NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

ALGORITHMS FOR EFFICIENT INTELLIGENCE COLLECTION

by

Duncan R. Ellis

September 2013

Thesis Advisor:	Nedialko B. Dimitrov
Second Reader:	Moshe Kress

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
3–9–2013	Master's Thesis	2011-07-05—2013-09-27

4. TITLE AND SUBTITLE

Algorithms for Efficient Intelligence Collection

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Duncan R. Ellis

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Postgraduate School
Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Department of the Navy

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited

13. SUPPLEMENTARY NOTES

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol Number: N/A

14. ABSTRACT

Modern intelligence techniques have drastically increased the rate at which communications data can be intercepted. The increased ability to collect and store this data poses a significant processing problem for intelligence agencies. We develop a software library, implementing a previously developed mathematical model of the information selection problem facing these agencies: given a time constraint, which items should be screened in order to maximize the relevant information obtained. Using our software, we analyze the performance of several screening strategies on a variety of representative intercepted intelligence networks, which we construct using real world data sets. We show the model consistently outperforms more naive approaches on networks with clusters of relevant sources, and highlight the importance of exploration in robust screening strategies.

15. SUBJECT TERMS

Analysis of algorithms, intelligence collection, graphical models, Bayesian inference, Markov random fields, networks

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			19b. TELEPHONE NUMBER *(include area code)*
Unclassified	Unclassified	Unclassified	UU	121	

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

ALGORITHMS FOR EFFICIENT INTELLIGENCE COLLECTION

Duncan R. Ellis
Lieutenant Commander, United States Navy
B.A., University of Iowa, 2003

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN OPERATIONS RESEARCH

from the

NAVAL POSTGRADUATE SCHOOL
September 2013

Author: Duncan R. Ellis

Approved by: Nedialko B. Dimitrov
Thesis Advisor

Moshe Kress
Second Reader

Robert F. Dell
Chair, Department of Operations Research

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Modern intelligence techniques have drastically increased the rate at which communications data can be intercepted. The increased ability to collect and store this data poses a significant processing problem for intelligence agencies. We develop a software library, implementing a previously developed mathematical model of the information selection problem facing these agencies: given a time constraint, which items should be screened in order to maximize the relevant information obtained. Using our software, we analyze the performance of several screening strategies on a variety of representative intercepted intelligence networks, which we construct using real world data sets. We show the model consistently outperforms more naive approaches on networks with clusters of relevant sources, and highlight the importance of exploration in robust screening strategies.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

List of Figures

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

API	Applied Programming Interface
bcc	blind carbon copy
cc	carbon copy
CSV	Comma Separated Value
FERC	Federal Energy Regulatory Commission
FHM	Finite Horizon MDP
IO	input and output
MDP	Markov Decision Process
PE	Pure Exploitation
SEC	U.S. Securities and Exchange Commission
SMS	Short Message Service
SQL	Structured Query Language
VDBE	Value-Difference-Based-Exploration
WEF	Wide Exploration First

THIS PAGE INTENTIONALLY LEFT BLANK

Executive Summary

Modern intelligence techniques have drastically increased the rate at which communications data can be intercepted for analysis. This increased ability to collect data, coupled with the growing use of cell phones, SMS messaging, and email as methods of information sharing, means collection agencies face a potentially overwhelming volume of intelligence data.

The intelligence cycle describes the process by which intelligence data is collected, processed, and evaluated. It consists of five stages (1) planning and direction, (2) collection, (3) processing, (4) analysis, and (5) dissemination. In this thesis, we focus on the processing stage, where an intelligence processor screens the data, considering the information's reliability, validity, and relevance. This processing stage often requires human involvement to forward relevant intelligence data to analysts, and is often time critical. The processor faces an information selection problem, and must decide which pieces of information to screen and in what order, to maximize the amount of useful data collected.

When deciding what pieces of information to screen, the processor faces a choice between exploiting sources that he already knows have provided useful information, and exploring to potentially uncover new sources. Often the time constraint is such that a processor might not have adequate time to screen every conversation or investigate every source. While many algorithms and heuristics currently exist to solve these types of exploration-exploitation problems, they assume independence among the sources, and might not be well suited to data with dependencies. In the context of intelligence collection, dependencies are likely, and even expected. Consider an intelligence processor faced with a source that is known to be relevant, and another, which is completely unknown. The presense of communications between the two might lead the processor to think the unknown source might also be relevant.

We implement a mathematical model to handle the information selection problem and develop a software library to allow for testing of different heuristic screening algorithms on a variety of intercepted intelligence network structures. The software consists of the following main components:

1. `GraphBuilder`: Uses the mathematical model, and is capable of reading in a large graph representing an intercepted intelligence network and constructing an object representing the knowledge of the processor. Methods are supplied which allow for updating of the processor's knowledge as items are screened. The software is capable of quickly updating

the probability distributions associated with maintaining the processor's current state of knowledge.

2. `MapBuilder`: Allows for the efficient generation of test networks representing intercepted intelligence networks from the Enron corpus, which contains the complete contents of 158 employee emails seized while the company was under investigation. Methods for data visualization, statistics collection, network trimming, and input and output (IO) are provided.
3. `Algorithms`: Contains heuristic algorithms for the screening optimization problem, as well as bounding selection methods representing best and worse case screening scenarios.

We use this software to conduct analysis on the mathematical model and screening algorithms. Key insights from the analysis are:

1. On graphs where relevant sources are clustered together the model consistently outperforms a simpler naive approach which does not account for dependencies. The model outperforms the naive approach by the largest margins when the intercepted intelligence network contains pockets of relevant sources surrounded by lower relevance noise. If the graph does not bear out the dependence assumptions, the model performs poorly.
2. Algorithms which place a high value on early exploration, such as Finite Horizon Markov Decision Process (FHM), offer the best performance across a wide range of graph structures and model parameters.
3. The model performs quite well even if the value of knowledge obtained from a known relevant source decreases over time.
4. Algorithm performance is highly dependent on the graph structure. Networks with a low density of relevant communications, where the relevant sources are not clustered together, have performance only slightly above a random selection method.

Acknowledgements

I wish to share my sincere appreciation to my advisor, Nedialko Dimitrov, for his guidance and support over the past year. I would also like to extend thanks to my second reader, Moshe Kress, for his helpful suggestions in refining this thesis. Finally, I express my gratitude to my wife, Anna, for her love and support during my time at NPS.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:
Background and Problem Description

1.1 Intelligence Processing

1.1.1 The Intelligence Cycle

The intelligence cycle, shown in Figure 1.1, describes the process by which intelligence data is collected, processed, and evaluated. It consists of five stages: planning and direction; collection; processing; analysis; and dissemination (Kaplan, 2012). In the *planning and direction* stage the specific intelligence requirements are identified. In the *collection* stage raw information is gathered from sources, which may be electronic, human, open source media, visual, or other. The *processing and exploitation* stage is the conversion of the raw information into finished intelligence. The processor screens the data, considering the information's reliability, validity, and relevance. In particular, data are screened such that only relevant items are considered for analysis. The processed information is analyzed in the *analysis* stage, converting the basic information into a finished intelligence product. The analyst puts the evaluated information in context and provides assessments suitable for decision makers. Finally, in the *Dissemination* stage, the processed information is collated into reports or other forms of communications and distributed to consumers, which may be either decision or policy makers.

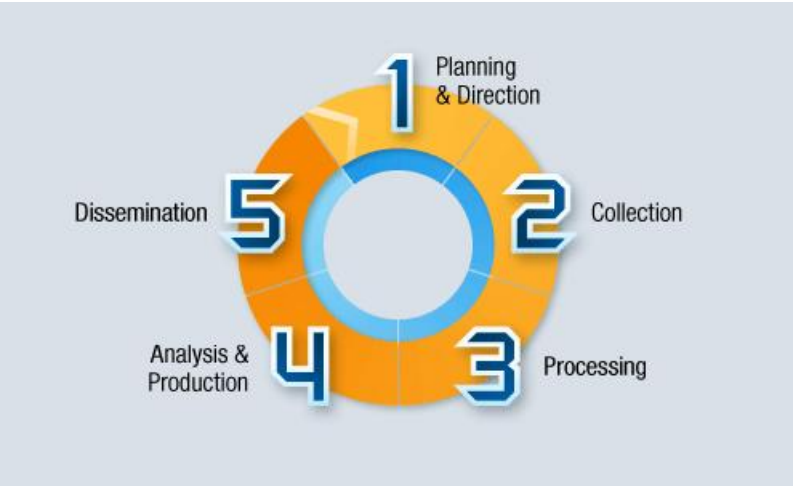


Figure 1.1: The intelligence cycle is the process of collecting and developing raw information into a finished product suitable for decision and policy makers and consists of five stages, which are listed in the figure. In this thesis, we focus on the Processing stage.

1.1.2 Information Overload

Modern intelligence collection technologies have drastically increased the rate at which communications data can be intercepted for analysis. This increased ability to collect data, coupled with the growing use of cell phones, Short Message Service (SMS) messaging, and email as methods of information sharing, means collection agencies face a potentially overwhelming volume of intelligence data (Hedley, 2007).

In this thesis, we focus on the processing stage, in which the operator, which we shall refer to as a processor, searches through and screens the data, using the results to aid in the preparation of the intelligence product. This processing stage often requires human involvement to forward relevant intelligence data to analysts. This stage is often also time critical; the processor must decide which pieces of information to screen and in what order, to maximize the amount of useful information collected within his time constraint. Faced with a potentially enormous volume of intelligence data, the processor might only have sufficient resources to screen a tiny percentage of the available data.

1.2 Prior Research and Similar Problems

1.2.1 Operations Research and Intelligence

The applications of operations research to intelligence problems is considered by Kaplan (2012) and is surprisingly limited. During the Cuban missile crisis of October 1962, the CIA retrospectively applied Bayes' rule to intelligence data to update the probability of Soviet missile shipments to Cuba (Zlotnik, 1967). Deitchman's Guerrilla model (Deitchman, 1962), followed by Schaffer (1968) addresses situational awareness, capturing information asymmetry between conventional and guerrilla forces. Atkinson and Wein (2010) develop models to locate terrorists in criminal networks by searching for criminal activities such as bank robberies or explosives procurement. Although other examples of intelligence research can be found in the literature, many focus on stage four, analysis and production, and do not address the question of information overload in the processing stage.

1.2.2 Ranking and Selection and Exploration/Exploitation

The problem of the processor has many similarities to traditional ranking and selection and exploration/exploitation problems. In ranking and selection, the problem can be defined as selecting the best alternative among a finite number of choices, where uncertainty exists in each alternative. While different methods are available to solve ranking and selection problems

(Fu et al., 2007), many do not address correlations between alternatives. Frazier et al. (2009) suggests a method to take correlations between alternatives into account, by using a knowledge gradient policy.

The processor faces a choice between exploiting sources that he already knows have provided useful information in the past, and exploring to potentially uncover new sources. Often, the time constraint is such that a processor might not have adequate time to screen every conversation or investigate every source. While many algorithms and heuristics currently exist to solve the exploration-exploitation problem (Berry and Fristedt, 1985), they assume independence among the sources, and might not be well suited to data with dependencies.

Dependencies are likely and indeed even expected in the context of intelligence collection. Consider a source A that the processor knows to be relevant. The presence of communications between A and another source B might lead the processor to think that B might also be a relevant source. These dependencies differentiate the intelligence collection problem from a typical ranking and selection or exploration-exploitation problem and might prove to be problematic if existing algorithms or heuristics are naively applied.

1.2.3 Information Selection in Intelligence Processing

In his master's thesis, Nevo (2011) considers a social communication network where a processor faces a pool of records, and must determine a screening strategy to maximize the number of relevant conversations obtained in a limited time period. He proposes a mathematical model utilizing methods from graphical models, social networks, random fields, and Bayesian learning to represent the knowledge of the processor. A summary of the problem setting and model can be found in Chapter II, with a complete description available in his thesis.

1.3 Chapter Outline

The thesis has six chapters. In Chapter II, we describe the mathematical model proposed by Nevo (2011) and describe a software tool based on this model. In Chapter III, we discuss methods of creating sample intercepted intelligence networks from the ENRON Corpus email database. Chapter IV discusses possible algorithms and heuristics to handle the information selection problem. In Chapter V we examine the performance of these algorithms and in Chapter VI we summarize the research and propose possible software modifications and model extensions suitable for future work.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 2:
Model Description and Software Implementation

In this chapter we formalize the problem setting and describe a mathematical model using techniques from graphical models, social networks, random fields and Bayesian learning. Finally, we describe the specific methodology and software implementation, which we use to test screening strategies.

2.1 The Model

2.1.1 Problem Setting

During the collection stage, intelligence data is intercepted from available sources, such as email, telephone conversations, and text messages. Each piece of data represents a conversation between two participants. The total of these intercepted conversations represents a network where the participants are nodes and an edge exists between nodes if they share at least one conversation, which we shall refer to as an item. This network is passed to the intelligence processor, along with a list of analysis objectives formulated by an intelligence analyst or agency, that the processor will use to assign a relevance value to any screened item. The processor must identify as many relevant items as possible in a given time period. This time period is generally not sufficient to screen the entire collection, and in some cases might only allow sufficient time to screen a very small percentage of the intercepted network. The processor therefore desires a screening strategy which maximizes the expected number of relevant items identified.

While items could have multiple levels of relevance depending on the provided intelligence objects, we consider a binary setting for simplicity - that is, an item is either *relevant* or *irrelevant*. Additionally, we consider the relevance of the participants, as information providers, to be measured on a discrete scale, for example *very low*, *low*, *medium*, *high*, and *very high*. The relevance values of two participants provides insight to the frequency of relevant items shared between them.

Prior to beginning the screening process, the processor is aware of the network topology, to include the number of available items between each pair of participants that are available for screening. The processor is also provided with some partial information about the network participants, enabling the establishment of an initial prior joint probability distribution for their

relevance values. The range of certainty the processor has about each participant's relevance may vary from complete uncertainty to absolute certainty. The processor also has some information from past screenings in the form of a conditional probability distribution concerning the probability of uncovering a relevant item between two participants if their relevance values are known.

The screening process proceeds in rounds. In each round, the processor selects an item for screening. The screening reveals the item as either relevant or irrelevant. In addition to the relevance of the item, the screening could also uncover relevance information about the participants, which we shall refer to as sudden revelation. These sudden revelations can occur in either relevant or irrelevant conversations, and serve to immediately identify with certainty the relevance value of a participant. We assume that the screening proceeds without error, so the relevance value of both the conversations and participants assigned by the processor represent their true relevance. The probability of screening a relevant conversation between two participants is a random variable whose probability distribution is updated in a Bayesian manner during the screening process. Each round reveals information that also allows the processor to update the probability distribution associated with the value of the participants on the screened edge.

2.1.2 Model Notation and Assumptions

We model the communications data the processor faces as a graph $G = (V, E)$. Each node represents a source with a discrete relevance value d_u. Each edge $(u, v) \in E$ represents a set of items between two participants that are available for screening. Let $q(e)$ be the subset of items for a single edge $e \in E$. Assuming independence, this subset of relevance items $q(e)$, forms a random sample from a Binomial distribution.

We model the probability that an item in the subset $q(e)$ is relevant as p_e, which is the parameter for the binomial distribution from which items in $q(e)$ are randomly drawn. The value of p_e is unknown to the processor. Although p_e is a continuous variable, with values $[0, 1]$, for model simplification we consider a set of discrete values. We model the probability that the value of d_u or d_v will be revealed while screening an item in $q(u, v)$ as an independent event for each of the two nodes, with a fixed probability c. If the values of $d_u, u \in V$, and $p_e, e \in E$, are known to the processor, along with the graph topology of G and the subsets $q(e), e \in E$, then the problem of the processor would be trivial - always screen an item from the edge e with the highest p_e. However, both the values of d_u and p_e are not known to the processor with certainly, rather are

represented by probability distributions which are updated during the screening process. Figure 2.1 shows a simple network between three participants where each edge has five items. The possible values of p_e and d_u are also given.

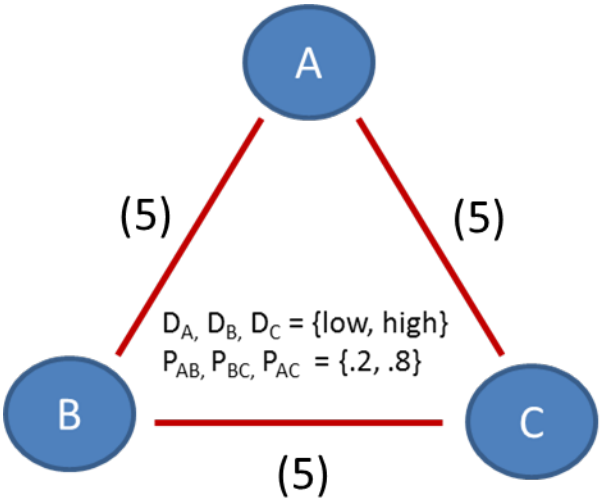


Figure 2.1: A graphical depiction of an intercepted intelligence network with three participants; A, B, and C, with possible discrete relevance values (d_u) of either high or low. Each pair of participants shares five items between them. The probability of an edge having a relevant item (p_e) is also discrete, with the values .2 or .8. Prior to beginning the screening process, the processor does not known the values of the d_u's or p_e's for any of the nodes or edges in the graph.

Since the values of p_e are unknown the processor, we use the random variable P_e to represent the processors belief of its value. Likewise, we let D_u represent the belief value of d_u, although unlike the value of p_e, the true value of d_u may be revealed to the processor during the screening process in the form of sudden revelation.

In addition to the graph topology and number of items in each edge, the processor begins the screening process with an initial prior distribution for $\overline{D}$, where $\overline{D} = (D_1, \cdots, D_{|V|})$. The Hammersley-Clifford theorem (Koller and Friedman, 2009) states this distribution can be specified as a product of potential functions on the maximal cliques of G. If the potential function $\Phi_C(\overline{D}_C)$ is given for all maximal cliques, then the distribution of $\overline{D}$ is the product of those potential functions. The processor is also provided a conditional probability distribution for P_e, given the relevance values of the participants are known. This conditional distribution is of the form $Pr[P_{uv} = p | D_u = d_u, D_v = d_v], u, v \in V$.

2.1.3 Updating Process

During the screening, the processor identifies items as either relevant or irrelevant, or perhaps observe some sudden revelation which will reveal the relevance value of a node. This information is used to update the processor's knowledge, represented in the model as the joint probability distribution of $[\overline{P},\overline{D}]$ denoted as $Pr[\overline{P},\overline{D}]$. With the random variables $\overline{D}$ forming a Markov random field and the assumption that the processor has a joint probability distribution of the relevance values of the participants and a conditional probability distribution for p_e, we specify the joint probability distribution for $Pr[\overline{P},\overline{D}]$ as

$$Pr[\overline{P},\overline{D}] = \frac{1}{Z}\prod_{C\in\mathscr{C}}\Phi_C[\overline{D}_C]\prod_{(u,v)\in E}Pr[P_{uv}|D_u,D_v] \tag{2.1}$$

where we let $\mathscr{C}$ represent the set of maximal cliques in G, and use Z as a normalizing constant. This joint probability distribution $Pr[\overline{P},\overline{D}]$ is updated during the screening process. We let $S_a = 1$ if an item on an edge is relevant, and 0 otherwise. Let $\overline{S} = (S_a, a \in q(u,v),(u,v) \in E)$. We form a new joint probability distribution $P[\overline{P},\overline{D},\overline{S}]$ including this additional knowledge as

$$Pr[\overline{P},\overline{D},\overline{S}] = \frac{1}{Z}\prod_{C\in\mathscr{C}}\Phi_C[\overline{D}_C]\prod_{(u,v)\in E}Pr[P_{uv}|D_u,D_v]\prod_{a\in q(u,v)}Pr[S_a|P_{uv}] \tag{2.2}$$

where $Pr[S_a|P_{uv}] = P_{uv}$ if $S_a = 1$, and $1 - P_{uv}$ otherwise. The updating process when the processor uncovers a relevant item can therefore be expressed as $Pr[\overline{P},\overline{D},\overline{S}|S_a = 1]$. If sudden revelation reveals the relevance value of a participant, we express the update as $Pr[\overline{P},\overline{D},\overline{S}|D_u = d]$ where d is the discrete relevance value, for example *low*, *medium*, or *high*.

2.2 Methodology

We use graphical models to represent the dependencies between the variables (Pearl, 1986). Factors for the joint probability distribution of $\overline{D}$, $\Phi_C[\overline{D}_C]$, are specified for every maximal clique in the graph. Factors are also specified to represent the conditional probability distributions for p_e, $Pr[P_{uv}|D_u,D_v]$. An example graphical model is shown in Figure 2.2 for the simple intelligence network of Figure 2.1 between three participants (A,B,C) which form a single clique of size three.

This clique is represented by the factor $\Phi_{\{A,B,C\}}[D_A,D_B,D_C]$, and its initial assumed distribution

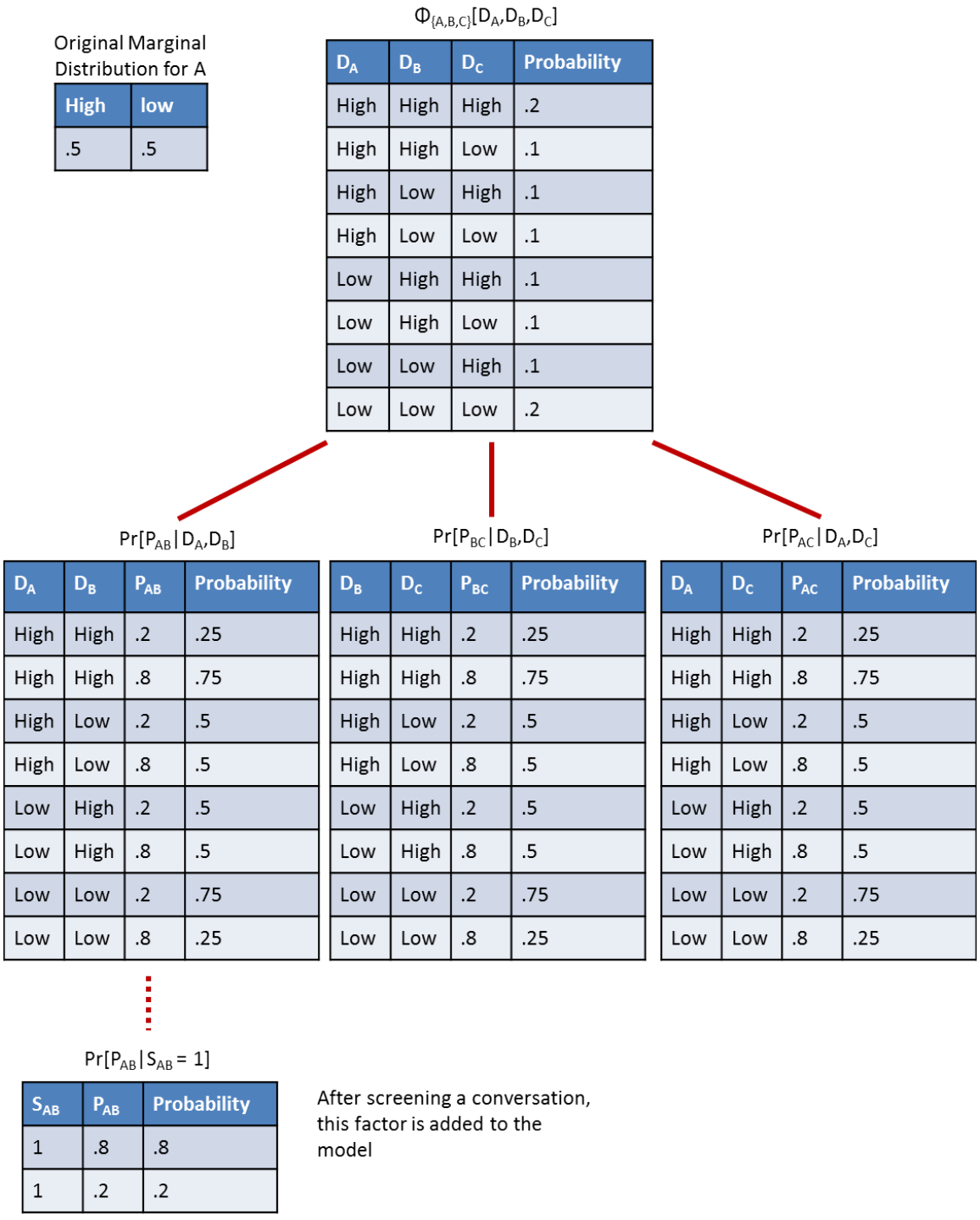


Figure 2.2: A graphical model for Figure 2.1 representing the knowledge of the processor. Factors $\Phi_{\{A,B,C\}}[D_A,D_B,D_C]$, $Pr[P_{AB}|D_A,D_B]$, $Pr[P_{BC}|D_B,D_C]$, and $Pr[P_{AC}|D_A,D_C]$ are specified to represent the joint distribution of the D_u's and the conditional probabilities of the P_e's. Edges (separators) are denoted by lines, and exist between factors if they share at least one variable. After screening a single item between A and B and finding it relevant, the factor $Pr[P_{AB}|S_{AB}=1]$ is added to the model, denoted by a dashed edge. The initial marginal distribution for D_A is also calculated by marginalizing $\Phi_{\{A,B,C\}}[D_A,D_B,D_C]$ and shown in the upper left.

is shown. Factors $Pr[P_{AB}|D_A,D_B]$, $Pr[P_{BC}|D_B,D_C]$, and $Pr[P_{AC}|D_A,D_C]$ represent the conditional probabilities for P_e for each edge. The initial marginal distribution for a particular D_u can be calculated by marginalizing $\Phi_{\{A,B,C\}}[D_A,D_B,D_B]$. In this initial distribution, $Pr[D_A = high]$ and the $Pr[D_A = low]$ are identical.

A sample update process is provided, and the resulting change in $\Phi_{\{A,B,C\}}[D_A,D_B,D_C]$ is shown in Figure 2.3. The processor screens a single item between participants A and B and determines that it is relevant to the intelligence query. To represent this process in the model a new factor of the form $Pr[P_{AB}|S_{AB} = 1]$ is introduced. The introduction of this factor can be seen in Figure 2.2.

$Pr[P_{AB}|D_A,D_B]$

D_A	D_B	P_{AB}	Probability
High	High	.2	.25*.2 = .05
High	High	.8	.75*.8 = .6
High	Low	.2	.5*.2 = .1
High	Low	.8	.5*.8 = .4
Low	High	.2	.5*.2 = .1
Low	High	.8	.5*.8 = .4
Low	Low	.2	.75*.2 = .15
Low	Low	.8	.25*.8 =.2

$\Phi_{\{A,B,C\}}[D_A,D_B,D_C]$

D_A	D_B	D_C	Probability
High	High	High	.2(.05 + .6) = .13
High	High	Low	.1(.05 + .6) = .065
High	Low	High	.1(.1 + .4) = .05
High	Low	Low	.1(.1 + .4) = .05
Low	High	High	.1(.1 + .4) = .05
Low	High	Low	.1(.1 + .4) = .05
Low	Low	High	.1(.15 + .2) = .035
Low	Low	Low	.2(.15 + .2) = .07

$Pr[P_{AB}]$

P_{AB}	Probability
.8	.8
.2	.2

Updated Marginal Distribution for A

High	low
.59	.41

Figure 2.3: The update process sums out the S_{AB} variable after the conversation is screened. The reduced $Pr[P_{AB}]$ factor is multiplied against the $Pr[P_{AB}|D_A,D_B]$ factor. Then, the resulting factor product is multiplied against $\Phi_{\{A,B,C\}}[D_A,D_B,D_C]$. Our updated marginal distribution for D_A (lower right) now shows we believe A more likely to be of high relevance than low.

Figure 2.3 shows the remainder of the update process, which happens when the S_{AB} variable is marginalized. First, the reduced $Pr[P_{AB}]$ factor is multiplied against the $Pr[P_{AB}|D_A,D_B]$ factor.

Then, the resulting factor product is multiplied against $\Phi_{\{A,B,C\}}[D_A, D_B, D_C]$. By this method, we update our prior distribution of $\overline{D}$. After normalization of our new $\Phi_{\{A,B,C\}}[D_A, D_B, D_C]$ we can calculate an updated marginal distribution for D_A. As is shown, after screening a single item between A and B, and finding it to be relevant, our belief about A is updated. The $Pr[D_A = high]$ now equals .59 and $Pr[D_A = low]$ equals .41. We now believe A is more likely to be of high relevance than low. The next section describes a software implementation of this graphical model structure.

2.3 Software Implementation

In this section we describe a software implementation of the above model and methodology. This software, which we shall refer to as `GraphBuilder`, is capable of reading in a large graph representing an intercepted intelligence network and creating an object that represents the knowledge of the processor regarding that network. Additionally, methods are supplied which update the processor's knowledge, either from the relevance value of a single screened item, or by sudden revelation of a participant's value. Finally, the software is capable of quickly calculating the joint probability distribution for $\overline{D}$, which yields the marginal distributions for any $D_U, U \in V$. The software builds on the `gPy` Python library developed by James Cussens at the University of York. [1] Complete *Applied Programming Interface (API)* documentation for `GraphBuilder` can be found in Appendix A.

2.3.1 Object Creation and Input Requirements

The `GraphBuilder` software creates an object that represents the knowledge of the processor. This knowledge is a collection of factors $\Phi_C[\overline{D}_C]$, specified for every maximal clique in an intercepted intelligence graph G. The knowledge also includes factors representing the conditional probability distributions for P_e. To construct these factors, `GraphBuilder` requires the following input parameters. Construction of these input parameters is discussed in detail in Chapter III.

1. A graph representing the intercepted intelligence network. Along with the physical topology that is known to the processor, node and edge attributes are also imported. Node attributes are the true relevance value of each participant. Edge attributes are the p_e values and the number of items available for screening. This graph structure represents the

[1] A complete description of the `gPy` library for graphical models can be found at the following site. Full documentation and a user manual are also provided. http://www-users.cs.york.ac.uk/jc/teaching/agm/gPy/

ground truth, and is used to assess the performance of screening strategies. Examples of intercepted intelligence networks can be found in Section 3.3.

2. A conditional probability table for P_e as described in Section 2.2. Table 3.2 provides an example of a conditional probability table.
3. Potential functions $\Phi_{|C|}$, for each maximal clique size in the graph. Table 3.3 provides an example for a maximal clique of size two.

2.3.2 Updating

As the processor screens items, the `GraphBuilder` object is updated to include the new knowledge gained, whether that knowledge is the relevance value of a screened conversation, or sudden revelation for a participant. Methods are provided to perform random draws for item screening and sudden revelation, and make subsequent edge and node updates to the `GraphBuilder` object.

1. Edge Updates: Two methods are provided for edge updates. The `random_draw()` method returns a random draw (either relevant or irrelevant) for an item on a requested edge, however doesn't write back the results of this screening to the `GraphBuilder` object. This random draw is weighted with the true value of p_e (which is unknown to the processor) for the edge requested. The `edge_update()` method allows the user to specify a relevance value for an item and updates the `GraphBuilder` object.
2. Node Updates: Two similar methods are provided for node updates. The `sudden_relevance_simple()` method returns the relevance value of a specified participant if sudden revelation occurs. This is a weighted draw, using the specified value of c (probability of sudden revelation) for the node. This method doesn't write back the results of any sudden revelation to the `GraphBuilder` object. The `node_update()` method allows the user to specify a relevance value for a participant and updates the `GraphBuilder` object.

2.3.3 Conditioning

In addition to the updating methods described in Section 2.3.2, `GraphBuilder` provides methods for calculation of the edges that have a high probability of returning a relevant conversation - i.e., a high $E[P_e]$ value. The methods build upon conditioning functions provided in the gPy library, which allow for the efficient calibration of a graphical model. Calibration ensures that all

factors associated with the cliques and separators [2] are the appropriate marginal distributions. The `highest_expected_pij()` method returns either an $E[P_e]$ value for a specified edge, or a sorted list of all $E[P_e]$ values for the entire graph. The `expected_di()` method returns the marginal distribution for a requested participant.

[2]Full documentation concerning gPy graphical model structure can be found at http://www-users.cs.york.ac.uk/jc/teaching/agm/gPy/Doc/API/

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 3:
Creating Sample Intelligence Networks

To facilitate testing of algorithms for intelligence collection, we desire the ability to construct test networks that are representative of real-world intercepted intelligence networks. These test networks must contain not only the topology known to the processor prior to beginning the screening process, but also the "ground truth" – that is the true values of $d_u, \forall u \in V$, and $p_e, \forall e \in E$, which we require to assess the performance of screening methods. We also desire the ability to create test networks with different topologies and d_u and p_e distributions to measure the effect of their variation. For example, we may wish to test the relationship between the variance of p_e and the effectiveness of a particular screening strategy.

We create a software tool, named `MapBuilder`, which allows for the efficient generation of test networks representative of real world intercepted intelligence networks from a real world data source, the Enron corpus. Additionally, methods for data visualization, statistics collection, and network trimming are provided. The capabilities of the `MapBuilder` tool are discussed in detail below, with complete API documentation provided in Appendix B for all referenced methods.

3.1 The Enron Corpus

In 2002, the Federal Energy Regulatory Commission (FERC) and U.S. Securities and Exchange Commission (SEC) publicly released a corpus of emails from 158 Enron employees to enable the public to better understand the motivations for their investigation of the company (Diesner and Carley, 2005). The corpus contains the contents of these 158 employee's email boxes over a time horizon of 3.5 years.[3] Diesner and Carley (2005) note that the corpus is of interest to researchers studying social networks, organizational behavior, and organizational theory as it enables the analysis of inter-company interactions over a multi year time horizon. For our purposes, the corpus is a rare example of a publicly available large communications network.

In Section 3.1.1 we describe a detailed procedure for transforming the raw corpus into a complete communications network representing the "ground-truth." In Section 3.3, methods for trimming the complete network to create intercepted intelligence networks are discussed.

[3]The complete corpus is available at http://www-2.cs.cmu.edu/ enron/

3.1.1 Creating the Complete Network

In its raw form, the Enron corpus contains 619,446 email messages contained in the mailboxes of 158 employees, with each separate email message stored as a text file. Although only 158 email boxes are contained in the corpus there are emails from 85,291 distinct email addresses, because many messages were either sent or received by participants outside the corpus.

To transform the raw corpus into a network we first import the data into a Structured Query Language (SQL) database for ease of manipulation using the `buildEnron()` method. Our database contains a single table, with each entry representing a conversation between two participants. We create "*from name*", "*to name*", "*to type*", and "*message text*" fields for each entry. Emails with multiple recipients, including carbon copy (cc) and blind carbon copy (bcc) recipients, are considered separate conversations and separate table entries are created for each pairing. For example, an email sent by participant A to participant B, with a cc sent to participant C, would generate two table entries; the first would be between A and B and the second between A and C. From the contents of each email, we concatenate the subject and message text and store it in the "*message text*" field. The expansion of the corpus in this manner yields a table that contains 3,065,082 emails between 85,291 distinct addresses.

We use the `buildGraph()` method to create the network directly from the SQL database. Each entry in the database table represents a single item between two participants. Keywords located in the "*message text*" field are used to define these items as either relevant or irrelevant to a particular intelligence query. For example, we might wish to denote every item that mentions "*New York*" or "*Washington*" as relevant.

An edge exists between nodes (participants) if they share at least one item between them. We record the number of relevant and irrelevant items on each edge and save these values in the network structure as edge attributes. We set the true p_e value for each edge as the proportion of the items on the edge that are relevant. We define the possible levels for the participant relevance values (d_u), for example *low*, *medium*, and *high*. We then calculate the d_u value for each node by sorting the nodes by the number of relevant items on their adjacent edges. We use a percentile function to divide the nodes into groups corresponding to the chosen discrete relevance values. This completes the creation of the complete network.

3.2 Data Summarization and Visualization

We provide methods to allow for the comparison of different networks created using the `MapBuilder` software. By tabulating network attribute statistics such as the number of relevant conversations, or edge p_e values, we summarize the differences between test networks. Additionally, we provide an efficient visualization schema for viewing larger networks that captures and highlights the features of the network.

3.2.1 Graph Statistics

The `graphStats()` method provides summary statistics for a network. The method calculates the number of nodes of each relevance value, and the total number of relevant and irrelevant items in the network. The largest maximal clique size and maximum node size (both by total and relevant items on its adjacent edges) are also calculated. Table 3.1 shows attributes of the complete Enron network created in Section 3.1.1 by `buildGraph()`. Items containing the words *New York*, *Washington*, or *California* are considered relevant in this example. We note that in this network only a very small percentage of items are relevant to the intelligence query.

Table 3.1: Summary statistics for the complete Enron network described in 3.1.1. Items with the keywords *New York*, *Washington*, or *California* are considered relevant. In addition to information provided in the table, `graphStats()` also calculates the largest maximal clique in this graph as containing 36 nodes. The largest node (sorted by total) has 106,985 items on its adjacent edges. The highest number of relevant items on edges adjacent to a node is 8,872.

	Relevance	*Count*	*Proportion*
Node	High	97	.00114
	Medium	228	.00267
	Low	84,966	.99619
Edge	Relevant	91,365	.02981
	Irrelevant	2,973,717	.97019

Two additional methods are provided which generate histograms for edge data. The `PEDist()` method plots a histogram of the edge P_e values, and also provides the ability to export the data to a text file. Figure 3.1 shows the distribution of P_e values for the complete Enron network, using the keywords from Section 3.2.1. The `conDist()` method plots a histogram for the number of relevant or total items available for screening on each edge. Figure 3.2 shows the distribution of the number of total items available for screening on each edge for the complete Enron network.

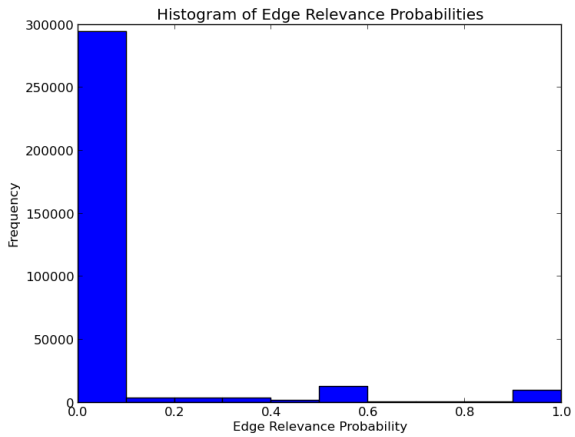


Figure 3.1: A distribution of the edge p_e values in the complete Enron network.

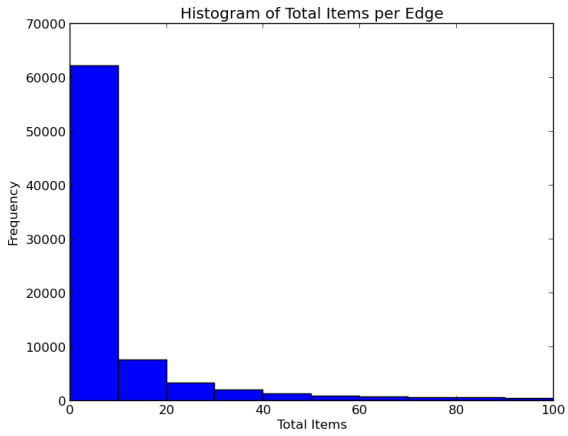


Figure 3.2: A histogram showing the distribution of the number of total items available for screening on the edges of the complete Enron network. The histogram is right censored at 100 items as the extremely long right tail makes visualization difficult. Edges with over 100,000 items are present in the network.

3.2.2 A Schema for Network Visualization

Many of the network structures we create are relatively large (greater than 200 nodes), and even summary information provided by `graphStats()` can mask certain structural characteristics. The `drawGraphRels()` method is capable of displaying large intelligence networks while capturing important structural attributes, such as node relevance, p_e values, and the location of maximal cliques. A complete description of `drawGraphRels()`, to include tuning parameters which allow for finer control over the default drawing parameters, is given in Appendix B.

Figure 3.3 provides an example `drawGraphRels()` output for a small network. We denote the

discrete relevance value of each node by its color. In Figure 3.3 nodes with *high* relevance are green, nodes with *medium* relevance are blue, and nodes with *low* relevance are red. The number and assignment of colors can be specified to customize the display. Node sizing is a function of the number of relevant items in their adjacent edges. The edge thickness is a linear function of the p_e values, with higher p_e edges having thicker lines than those with low p_e values.

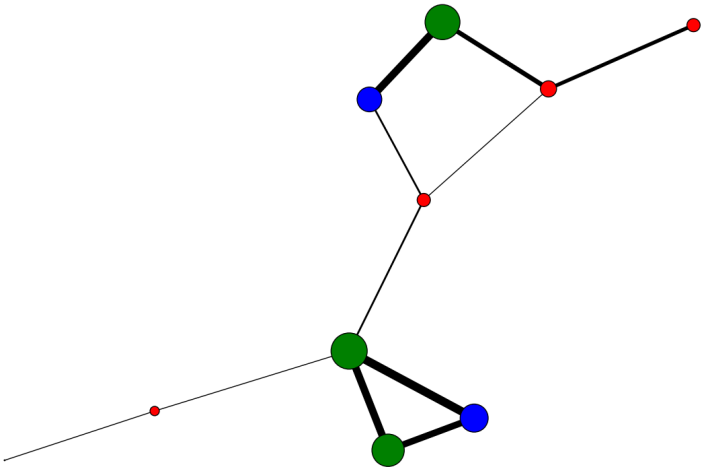


Figure 3.3: A small intelligence network with 10 participants. Three of the participants have a *high* relevance value, and are green. Two participants are of *medium* relevance and are blue, and the remaining participants are of *low* relevance and are red. The larger the node size, the more relevant items are contained in its adjacent edges. Edges with higher thickness have higher p_e values, denoting the probability of screening a relevant item on these edges is higher.

3.3 Building Intercepted Intelligence Networks

The size of the complete Enron communications network makes it impractical for testing screening techniques, as the time to update the processor's knowledge would be prohibitively long. In order to conduct efficient testing, we require the ability to conduct multiple runs of each algorithm over several hundred iterations while still maintaining reasonable run times.

In this section, we discuss some methods for creating smaller intercepted communications networks, which we shall refer to as sub-graphs, from the complete network. This sub-graphs are created in a manner such that they are still representative of real-world communications networks. We propose three basic network trimming techniques using the methods `trimGraphDeep()`, `trimGraphWide()`, and `trimGraphInfection()`. Complete API documentation is provided in Appendix B. We intend these methods to approximate methodologies a real world agency might use during the collection stage.

We further separate these trimming methodologies into *targeted* and *naive* versions. In a naive collection method, the collection agency has no prior information concerning the relevance of participants in the complete network. In the targeted version, there exists some partial information that allows the agency to better focus their collection efforts, particularly in determining the initial nodes to add to the sub-graph.

3.3.1 The Deep Method

The first trimming method we propose is `trimGraphDeep()`, a method for creating intercepted intelligence network sub-graphs using what we refer to as a *deep method*. We first consider a targeted version of this methodology, in which the intelligence agency has some prior information concerning the relevance values of participants in the complete network.

We begin by identifying a specified number of participants in the complete Enron network with the highest relevance values. We think of this step as the collection agency having targeted intelligence on the most likely suspects. We then add all neighbors of these targeted participants to the sub-graph. The remainder of the sub-graph creation method proceeds for a specified number of rounds.

In subsequent rounds, the node with the highest relevance value is identified from the neighbors added during the previous round, and its neighbors are added to the sub-graph. We refer to this method as the deep method as the collector is only considering candidates for the next node of maximum relevance from the last group of neighbors added to the sub-graph, going as deep into the network as the number of rounds permits.

Even with limited rounds, the size of the sub-graphs created with this technique are generally too large to be processed by the `GraphBuilder` software. Using the relevance keywords *California* and *Washington*, a sub-graph created by `trimGraphDeep()` with only three rounds has 1,723 nodes. To reduce the sub-graph to a more manageable size, we apply a method of probabilistic pruning, removing all degree one nodes with a specified probability p. With a pruning probability of $p = .9$, three rounds of `trimGraphDeep()` produces a sub-graph of approximately 200 nodes, a significant reduction in size.

In addition to the targeted method, we also consider a naive deep method. The rounds proceed as in the targeted version, however instead of adding the neighbors of the node with the highest relevance value, we add the neighbors of the node with the highest number of total items (both relevant and irrelevant items) on its adjacent edges.

Sub-graphs constructed using the `trimGraphDeep()` method might be similar to intercepted intelligence networks created by phone tapping. An initial participant's phone, chosen on prior information concerning the participant's relevance, is tapped, and all conversations between that participant and second parties are recorded. From those second parties, either further targeted intelligence or simply call volume leads to the next phone to be tapped, and the collection continues. Figure 3.4 shows a visual representation for a graph created by `trimGraphDeep()`. The targeted method was used, with one initial node, three rounds of screening, and all degree one nodes pruned with probability .9. Figure 3.5 shows summary statistics for the graph in Figure 3.4 and a similar graph constructed with the naive version of `trimGraphDeep()`.

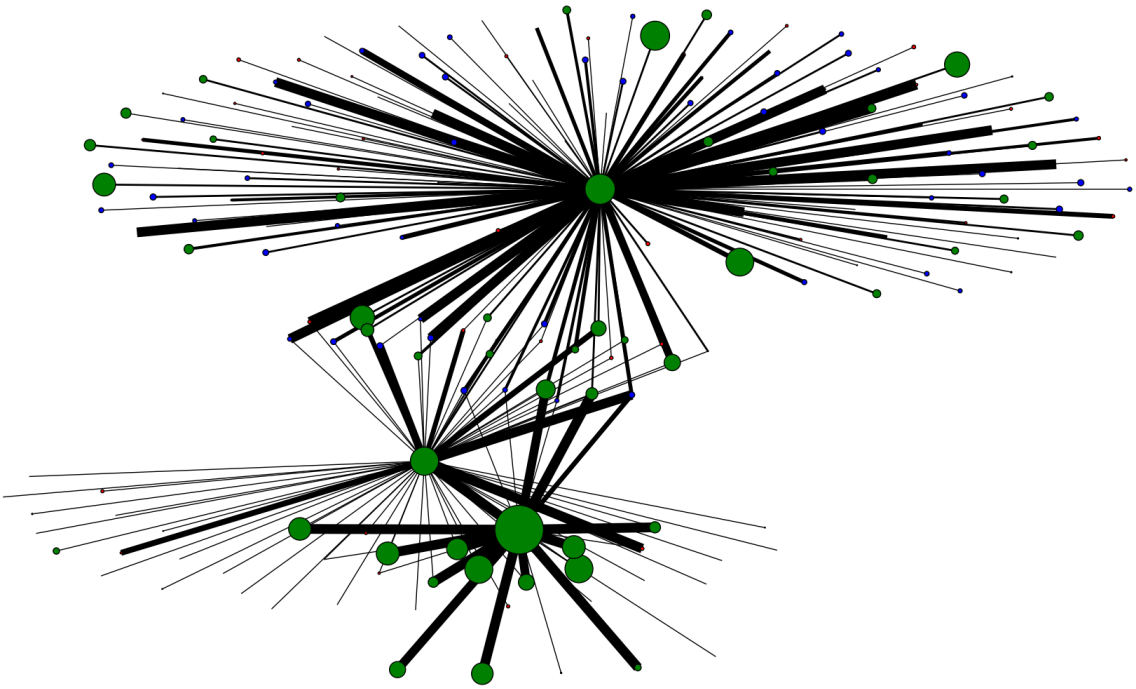


Figure 3.4: A sub-graph representing an intercepted intelligence network created with the targeted version of `trimGraphDeep()`. Three rounds of screening, one initial node, and all degree one nodes pruned with probability .9 are used as input parameters.

3.3.2 The Wide Method

Our next trimming method is `trimGraphWide()`, a method for creating sub-graphs using a *wide method*, which is similar in its basic structure to the *deep method* described in Section 3.3.1. Similar to `trimGraphDeep()`, the method proceeds for a specified number of rounds before termination. We consider a targeted version where initial nodes added to the sub-graph

are determined by selecting the participants with the highest relevance values. We then add all neighbors of these targeted participants to the sub-graph.

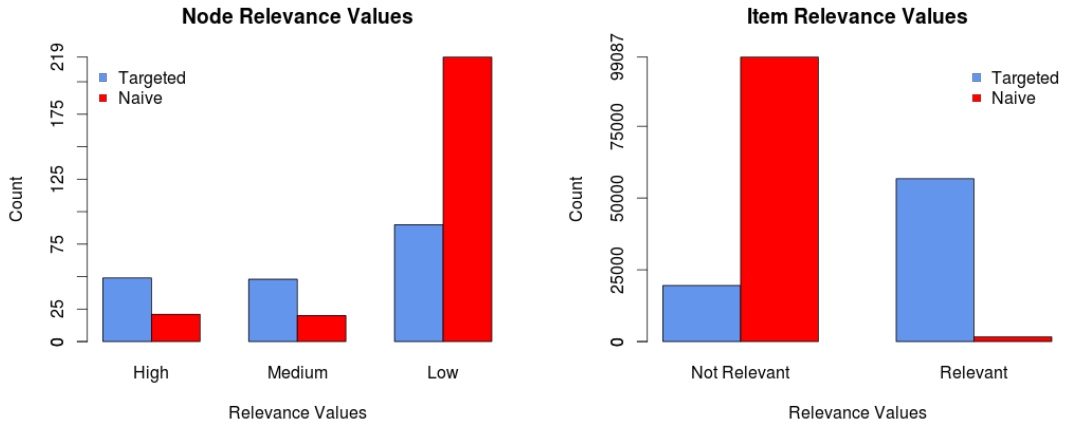


Figure 3.5: Statistics for sub-graphs representing an intercepted intelligence network created with the targeted and naive versions of `trimGraphDeep()`. Three rounds of screening, one initial node, and all degree one nodes pruned with probability .9 are used as input parameters. For the targeted version, the largest maximal clique in the graph has 3 nodes. The largest node (sorted by total items) has 84,944 items in its adjacent edges. The highest number of relevant items in edges adjacent to a node is 64,256. For the naive version, the largest maximal clique in the graph also has 3 nodes. The largest node (sorted by total items) has 106,999 items in its adjacent edges. The highest number of relevant items in edges adjacent to a node is 5,566.

In subsequent rounds, we add nodes with a slightly different strategy than `trimGraphDeep()`. Rather than consider candidates for the next node of maximum relevance only from the group of neighbors added to the sub-graph in the previous round, we consider ALL nodes previously added to the sub-graph. We refer to this method as the *wide method* because the collector is considering candidates from a larger group than in the *deep method*. This method is slightly more computationally expensive, as every round we must calculate a sorted list of node relevance values for a sub-graph size of increasing size. After the specified number of rounds is completed, the graph is probabilistically pruned, removing all degree one nodes with p. For a given number of rounds, `trimGraphWide()` method produces similar sized graphs as `trimGraphDeep()`.

Figure 3.6 shows a visual representation for a graph created by `trimGraphWide()`. The targeted method was used, with one initial node, three rounds of screening, and all degree one nodes pruned with probability .75. Figure 3.7 shows summary statistics for the graph in Figure 3.6

and a similar graph constructed with the naive version of `trimGraphWide()`.

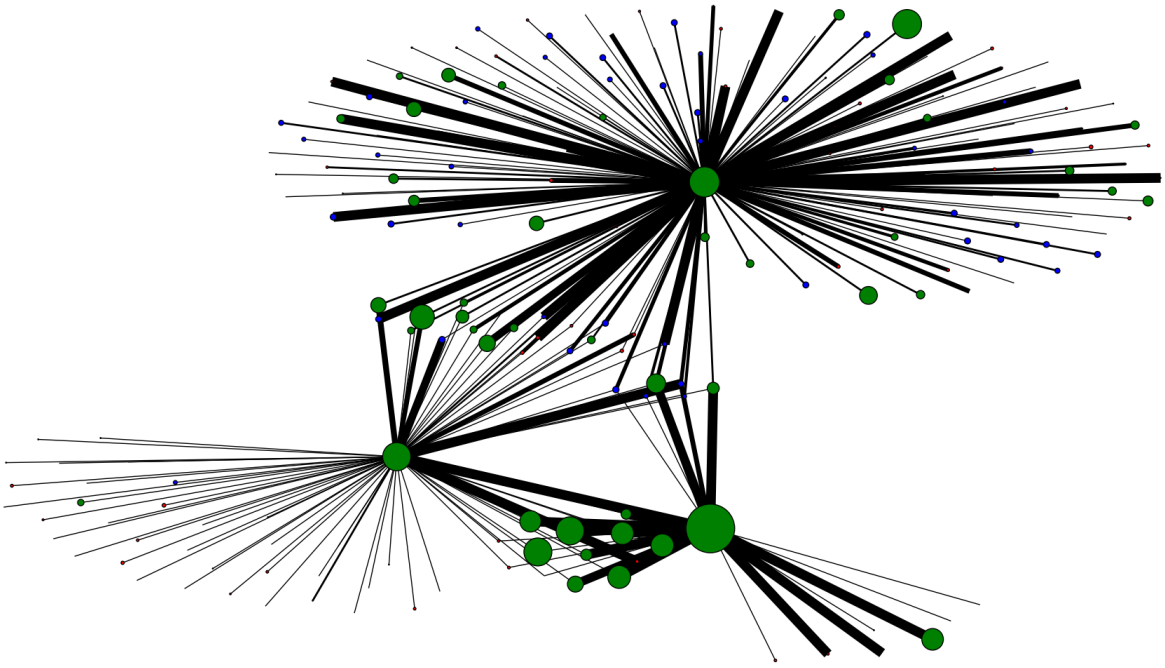


Figure 3.6: A sub-graph representing an intercepted intelligence network created with `trimGraphWide()`. Three rounds of screening, one initial node, and all degree nodes pruned probability .75 are used as input parameters.

3.3.3 The Infection Method

Our final method of sub-graph creation is quite different from the methods described in Sections 3.3.1 and 3.3.2. The `trimGraphInfection()` method attempts to simulate results from collection methods used in the interception of wireless signals. In this case, we assume the collector is only able to intercept and record a proportion of items (signals) emitted or received by a participant, where as in `trimGraphDeep()` and `trimGraphWide()` we intercepted all of them.

The screening process proceeds for a specified number of rounds. In the targeted version, we begin by identifying a specified number of participants with the highest relevance values, and add them to the sub-graph. During each round, edges adjacent to nodes already existing in the sub-graph are added with probability p, which we shall refer to as the infection probability. The naive version differs only in that the initial participants are added to the sub-graph based

on the total number of items on their adjacent edges, rather than relevant items. This *infection method* is more likely to add nodes to the sub-graph with high degree, as those nodes have more adjacent edges, and subsequently a higher probability of infection. Figure 3.8 shows a visual representation for a graph created by `trimGraphInfection()`. The targeted method is used, with an upper bound of 200 nodes and an infection probability of .001. Figure 3.9 shows summary statistics for the graph in Figure 3.8 and a similar graph constructed with the naive version of `trimGraphInfection()`.

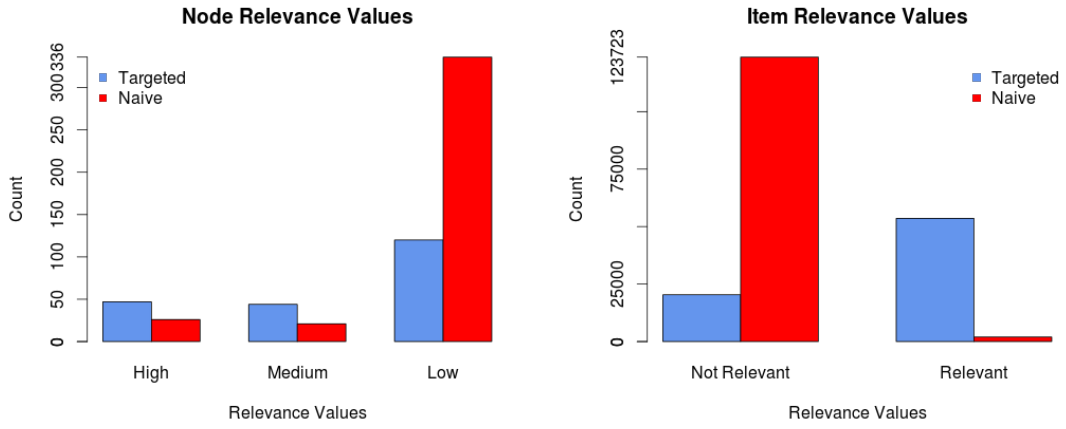


Figure 3.7: Statistics for sub-graphs representing an intercepted intelligence network created with the targeted and naive version of `trimGraphWide()`. Three rounds of screening, one initial node, and all degree nodes pruned with probability .75 are used as input parameters. For the targeted version, the largest maximal clique in the graph has 3 nodes. The largest node (sorted by total items) has 84,944 items in its adjacent edges. The highest number of relevant items in edges adjacent to a node is 64,256. For the naive version, the largest maximal clique in the graph has 4 nodes. The largest node (sorted by total items) has 106,999 items in its adjacent edges. The highest number of relevant items in edges adjacent to a node is 8,268.

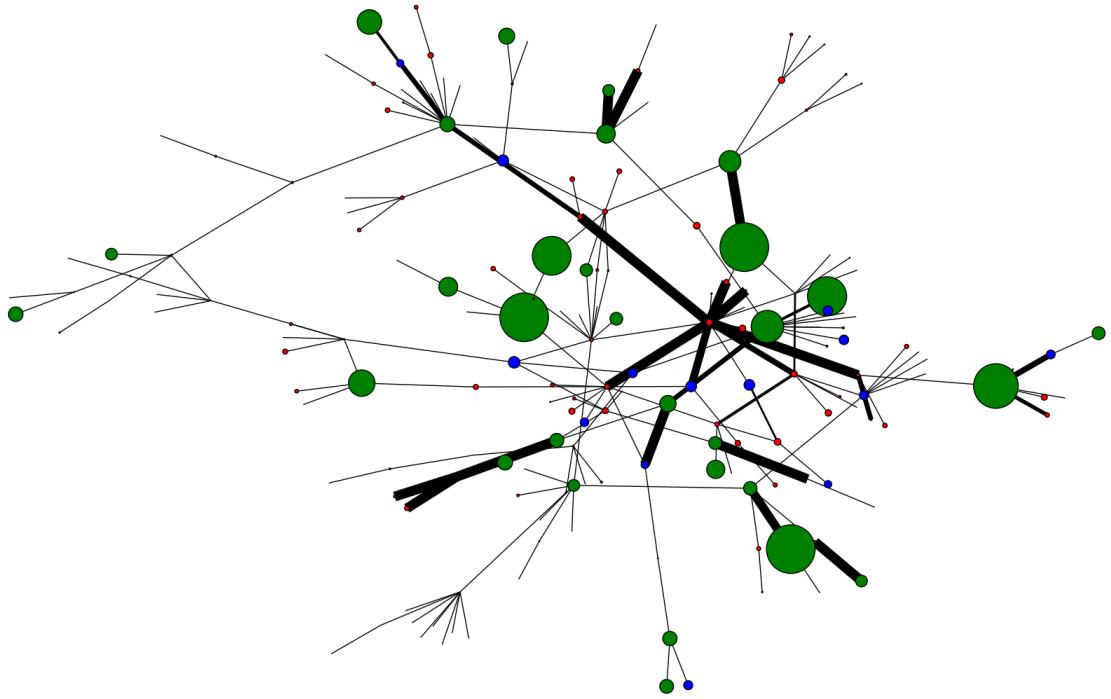


Figure 3.8: A sub-graph representing an intercepted intelligence network created with `trimGraphInfection()`. 184 nodes and an infection probability of .001 are used as input parameters.

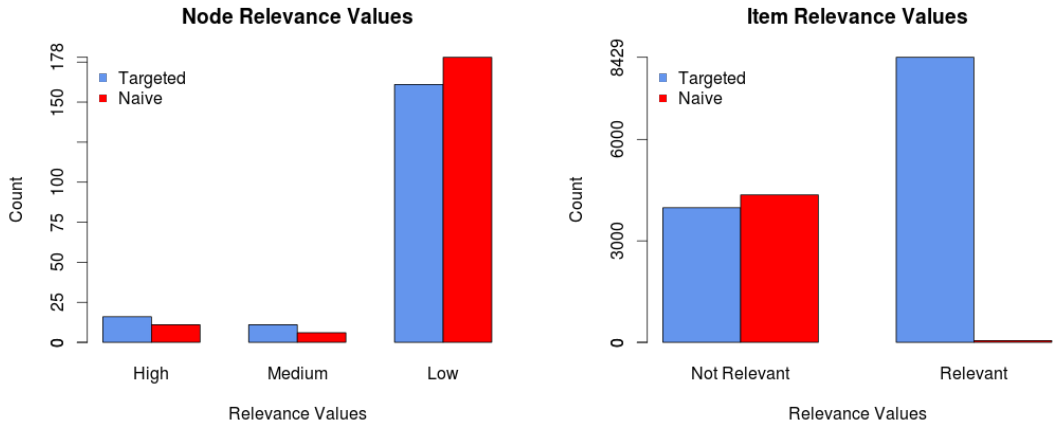


Figure 3.9: Statistics for sub-graphs representing an intercepted intelligence network created with the targeted and naive versions of `trimGraphInfection()`. An upper bound of 200 nodes and an infection probability of .001 are used as input parameters. For the targeted version, the largest maximal clique in the graph has 3 nodes. The largest node (sorted by total items) has 84,944 items in its adjacent edges. The highest number of relevant items in edges adjacent to a node is 64,256. For the naive version, the largest maximal clique in the graph has 2 nodes. The largest node (sorted by total items) has 106,999 items in its adjacent edges. The highest number of relevant items in edges adjacent to a node is 1,994.

3.4 Building Prior Distributions and Conditional Distributions

In Section 2.1.2 we discuss the requirement that the processor has an initial prior joint distribution of the node relevance values, $\overline{D}$, and a conditional probability of the form $Pr[P_{uv} = p|D_u = d_u, D_v = d_v], u, v \in V$. In a real world setting, the processor might generate these distributions from analysis of previous intelligence data or by consulting with subject matter experts. To establish reasonable distributions for testing we again consider the Enron corpus network, generating our prior distributions of $\overline{D}$ and conditional distributions for p_e directly from the data. We can think of the networks we create as being similar to a repository of past analysis where the processor is able to see both the true participant relevance values $\forall V \in G$, and the true p_e values, $\forall E \in G$. We provide methods in *MapBuilder* to generate both the initial prior distribution of $\overline{D}$ and the conditional distribution for p_e from Enron network data.

3.4.1 Building the Conditional Distribution for p_e

The `create_pij_dij_csv()` method creates a conditional probability table for $Pr[P_{uv} = p|D_u = d_u, D_v = d_v], u, v \in V$. We use a two step method to create the table. In the first step, we iterate through the edges of the graph, and sort the true p_e values into bins determined by the relevance of their adjacent nodes. For example, we locate all p_e values in the graph where both adjacent nodes have *high* relevance values, and place those p_e values in a bin. These bins represent a discrete probability distribution of the true p_e values, conditional on the node relevance values.

In the second step, we use a step function to further sort each bin of p_e values into sub-bins, where each sub-bin is a discrete p_e level specified as a parameter to the `create_pij_dij_csv()` method. Table 3.2 shows sample output for a conditional probability table with two node relevance values and two p_e levels. We note that in this example, knowing both participants have a *high* relevance value leads us to estimate the probability the p_e value is .75 as twice as likely than in the case where both participants have *low* relevance values. The conditional probability tables created by `create_pij_dij_csv()` are written to Comma Separated Value (CSV) files which can be imported by a `GraphBuilder` object when we create our graphical model.

3.4.2 Building the Prior Distribution

The `create_di_csv()` method is used to build tables for the prior joint distribution of $\overline{D}$ using similar techniques as `create_pij_dij_csv()` in Section 3.4. We specify a prior joint distribution for the values of d_u for every maximal clique size in the graph, and write each one to

a separate CSV file suitable for importing into `GraphBuilder` during creation of the graphical model. For example, in a graph that contains maximal cliques of two and three nodes, we would create a prior joint probability distribution for $Pr[D_u = d_u, D_v = d_v], u, v \in V$ and $Pr[D_i = d_i, D_u = d_u, D_v = d_v], i, u, v \in V$.

Table 3.2: A conditional probability table for $Pr[P_{uv} = p | D_u = d_u, D_v = d_v]$ created by `create_pij_dij()` with two node relevance levels and two p_e levels. We note that in this case, knowing both participants have a *high* relevance value leads us to estimate the probability the p_e value is .75 as twice as likely than in the case where both participants have *low* relevance values.

Node Relevance	Node Relevance	p_e level	Probability
high	high	.25	.94805
high	high	.75	.05195
low	high	.25	.95163
low	high	.75	.04839
low	low	.25	.97583
low	low	.75	.02427

To construct the prior distributions, we first separate the graph into its maximal cliques, and then group these cliques by their size. Each clique in the graph has an associated set of node relevance values. For example, a clique of size two might have one node with *high* relevance, and one node with *medium* relevance. For each clique size, we record the frequency that each node relevance set occurs, and use the resulting frequencies to construct a prior joint probability distribution for the clique. Consider a graph with two cliques; the first clique has two *high* relevance nodes and the second clique has two *low* relevance nodes. Our prior joint distribution for $\overline{D}$ would be $Pr[D_u = high, D_v = high] = .5$ and $Pr[D_u = low, D_v = low] = .5$. Table 3.3 shows a sample joint probability distribution created for a network's maximal cliques of size two.

Table 3.3: A prior joint probability distribution created by `create_di_csv()` for a network's maximal cliques of size two.

Node Relevance	Node Relevance	Probability
high	high	.03680
high	low	.17296
low	high	.17296
low	low	.61725

3.5 Input and Output

We provide input and output functions in *MapBuilder* to allow for more convenience in working with sub-graphs created from the Enron network. The `writeGraph_CSV()` method writes a graph to a CSV file and `readGraph_CSV()` reads in a CSV file from a previously saved graph. By providing these two functions, we allow for graphs to be created and stored for further use and analysis by the `GraphBuilder` software.

CHAPTER 4:
Algorithms

In this chapter we describe the `Algorithms` module, which contains heuristic algorithms for the screening optimization problem, as well as *bounding methods* representing best and worst case screening scenarios. Full API documentation for the `Algorithms` module can be found in Appendix C. Parameter tuning and the performance of these algorithms on networks created using techniques in Chapter III, is discussed in Chapter V.

4.1 Algorithm Performance Statistics

In order to compare algorithm performance, we establish a set of common statistics. Our first statistic is the number of relevant items identified by the algorithm in a specified number of iterations. This is our principal metric for performance, as our goal is to maximize the amount of relevant data the processor obtains during a limited screening time. Additionally, for each screened edge, we record the difference

$$\max_{e}\{p_e\} - p_{e^*} \tag{4.1}$$

where e^* is the edge screened by the algorithm. This is simply the distance between the p_e value of the optimal edge (highest p_e valued edge with items available for screening) and the p_e value of the chosen edge. Finally, we return the total run-time and the average iteration run-time.

4.2 The Value of Knowledge

Each iteration of an algorithm results in the identification of either a relevant or irrelevant item. We assign a value to this item representing the knowledge it provides to the processor. By default, this value is set to one for a relevant item, and zero if the item is irrelevant, however we provide the ability to substitute a function with any number of parameters. For example, we might wish to set the value of the first relevant item identified on an edge higher than subsequent relevant items. This is reasonable, as we might expect subsequent relevant conversations on that edge to contain duplicate information. Specific knowledge reduction functions, and their impact on algorithm performance are discussed in Section 5.7.

4.3 Bounding the Performance

To better understand the performance of an algorithm we create bounding selection methods representing best and worst cases for performance. We provide a *perfect* selection method in Section 4.3.2 that apriori knows the p_e values as an upper bound for algorithm performance, and a *random* selection method in Section 4.3.1 as a lower bound.

4.3.1 The Random Method

To establish a lower bound, or worst case scenario for the performance of any employed screening strategy, we provide the method `randompick()`, which implements a random selection method. In this scenario the processor is memoryless, begins screening with no prior distribution for $\overline{D}$ or conditional distribution for P_e, and has knowledge only of the network topology. Unable to accumulate knowledge from prior screenings, `randompick()` simply picks a uniformly random edge with available unscreened items.

4.3.2 The Perfect Method

The upper bound for the performance of a screening strategy is the case where the processor has perfect knowledge. If the processor knows the true values of p_e for every edge in the network, then the optimal selection process is a simple greedy heuristic; screen an item from the set of available edges with the highest p_e value. We implement this strategy in the `perfect()` method.

4.4 Pure Exploitation

We implement a greedy Pure Exploitation (PE) algorithm in the `PE()` method. This simple algorithm selects the next item for screening from the edge with the highest $E[P_e]$ value, that is, the edge with the highest expected probability of containing a relevant item. This algorithm performs no exploration, however it is useful as a benchmark against more sophistical screening strategies. The Pure Exploitation strategy is optimal if $Var[P_e] = 0, \forall e \in E$. We note that although the edge selection strategy in Pure Exploitation is not complex, the algorithm is still dependent on the non-trivial task of updating the processor's knowledge state after each round.

4.5 Softmax

The Softmax algorithm implements a mixed strategy of exploration and exploitation. (Thrun, 1992). The algorithm assigns a weight w_e between zero and one to each edge, where w_e is

the probability an item on edge e will be selected for screening. Weights are assigned using a Bolzman distribution

$$w_e = \frac{\exp(\frac{v_e}{K})}{\sum_e \exp(\frac{v_e}{K})} \tag{4.2}$$

where $v_e = E[P_e]$ and K is a tuning parameter often referred to as temperature (Daw et al., 2006). For small values of K, the weight of edges with large $E[P_e]$ values is high and items on those edges are more likely to be chosen. This is an exploitation dominated strategy. For large values of K, all edges have similar weights and random exploration dominates. We implement this algorithm in the `softmax()` method.

4.6 VBDE

The Value-Difference-Based-Exploration (VDBE) algorithm, introduced by Tokic and Palm (2011) mixes exploration and exploitation probabilistically using a modification of an ε-greedy algorithm, and is implemented in the `VDBE()` method. In each iteration, the algorithm assigns a probability ε that exploration is chosen. When there is a low certainty regarding the expected value of alternative actions the algorithm explores, exploiting otherwise. The value of the exploration likelihood, ε, is initially set to 1 and updated at each iteration using the formula

$$\varepsilon^{k+1} = \delta \frac{1 - e^{\frac{-U}{\sigma}}}{1 + e^{\frac{-U}{\sigma}}} + (1 - \delta)\varepsilon^k, \tag{4.3}$$

where $U = \max_{uv} |E^k[P_{uv}] - E^{k-1}[P_{uv}]|$, the maximum difference in expectations between the $(k-1)$st screening and the kth screening. The *inverse sensitivity* parameter σ determines the immediate impact a certain change in expectation has on ε. The δ parameter determines the *decay* rate of ε when the system is stable, that is, when there are very few changes in the $E[P_e]$ values.

During exploration iterations VDBE uses the Softmax algorithm with a relatively high temperature (K) value. The algorithm defaults to $K = .25$, however this parameter can be specified. For exploitation iterations, the Pure Exploitation algorithm is used.

4.7 WEF

The Wide Exploration First (WEF) is a simple heuristic that combines a wide exploration first policy with the Softmax algorithm, and is implemented by the `WEF()` method. A number of exploration iterations is specified as an algorithm parameter, along with an exploration parameter B. During the exploration phase, we select the edge with the highest $E[P_e]$ value, as long as it has been chosen less than B times. With this policy, the smaller the value of B, the more edges the algorithm will explore, although its exploration choices are never random. In the exploitation phase we pick edges using Softmax with a specified temperature parameter K.

4.8 Finite Horizon MDP

Our final algorithm is a finite horizon implementation of a Markov Decision Process (MDP), implemented by the `FHM()` method. The Finite Horizon MDP (FHM) algorithm can be thought of as a type of Knowledge-Gradient policy (Frazier et al., 2009), where the decision maker chooses at each iteration the alternative with the highest expected change in value. In our case, the value of a particular state is only known with certainty at the final iteration (time T), so an exact approach must look T rounds into the future to compute the best alternative. This results in an extremely prohibitive run time of $O(|E|^T \cdot \text{Infer})$, where Infer is the time required to update the knowledge state of the processor. We therefore implement the FHM algorithm using an estimate of the state value at a determined depth as a trade off between optimality and speed.

We begin by defining a `ChoiceNode` object, which holds the knowledge of the processor (h_i) in round i, with $T - i$ rounds remaining. The `ChoiceNode` object has a single method `getVal()` which returns the alternative with the highest value. The `ChoiceNode` value is calculated in three ways:

1. If the rounds remaining equals zero (the final iteration), there is no additional value to be gained, and the `getVal()` method returns 0.
2. If the depth equals zero, then we return an estimate of the states' value, assuming that no more belief distribution updates are performed. This value is

$$\sum_{a \in A} E[P_{e_a}] \tag{4.4}$$

where A is the set of the $T - i$ most likely relevant items under $Pr[\overline{P}, \overline{D} | h_i]$, and e_a is the

edge of item $a \in A$.

3. If the depth is greater than zero, then we create an object of type `RandomNode` for each available choice (edge with available item for screening), and call its `getVal()` method. The `ChoiceNode` then returns the max value from the `RandomNode` `getVal()` calls.

The value of the `RandomNode` is calculated as an expectation over all possible values of the choice. This expectation is taken pretending that the belief distribution of the parent `ChoiceNode` is the truth. This is because the processor only knows that particular belief distribution, and while he can hypothesize how it might change in the future, he does not know the true values of the parameters. For example, consider a simple model where the probability of sudden revelation (c) equals zero. In this case there are only two possible outcomes of the screening choice, either the screened item is relevant or the screened item is irrelevant. Since we also have to take into account the additional value of choices in future screening decisions, we create an updated `ChoiceNode` for the two states of knowledge, one where the item is relevant, and one where it's irrelevant, and call their `getVal()` methods.

Figure 4.1 shows a partial example of a single iteration of `FHM()` for the simple intelligence network of Figure 2.1 between three participants (A, B, C), with possible node relevance values of *high* or *low*. `FHM()` starts by creating a `ChoiceNode` object and calling its `getVal()` method, denoted by the square box at the top of the figure. Since the depth is greater than zero, we create a `RandomNode` object for each of the three edges and call their `getVal()` methods. `RandomNode` objects are denoted by circles. The `getVal()` process for the `RandomNode` created by the (A,B) choice is shown. There are 18 possible values that can result from choosing edge (A,B), shown in Table 4.1, and Figure 4.1 enumerates four of them. For each of the 18 possible values of the (A,B) choice, a new `ChoiceNode` object is created at the depth zero level and the `getVal()` method of each `ChoiceNode` returns an estimated value. The `RandomNode` then returns its value as an expectation over all values of the depth equals zero ChoiceNodes. This process is also completed for the (B,C) and (A,C) choices, however this is not shown in the figure. The value of the top `ChoiceNode` is then calculated as the max value from the set of children `RandomNodes`, {(A,B), (B,C), (A,C)}. The edge associated with this value is selected as the next edge for screening.

Table 4.1: The 18 possible outcomes that can result when an edge in a graph with two relevance levels is chosen. The outcomes are combinations of the conversation being relevant or irrelevant, and whether the value of the nodes are revealed or not by sudden revelation.

Item Relevance	*Node One Relevance*	*Node Two Relevance*
0	high	high
0	high	low
0	high	not revealed
0	low	high
0	low	low
0	low	not revealed
0	not revealed	high
0	not revealed	low
0	not revealed	not revealed
1	high	high
1	high	low
1	high	not revealed
1	low	high
1	low	low
1	low	not revealed
1	not revealed	high
1	not revealed	low
1	not revealed	not revealed

As the number of choices available on larger graphs can result in prohibitively long run times, we provide the ability to limit the number of edges that each `ChoiceNode` considers. We implement this restriction with a user provided integer parameter that specifies the number of `RandomNode` objects to create. With a limit specified, the `ChoiceNode` object will create half the `RandomNode` objects from the edges with the highest $E[P_e]$ values, and the other half by selecting uniformly random edges from the remaining choices.

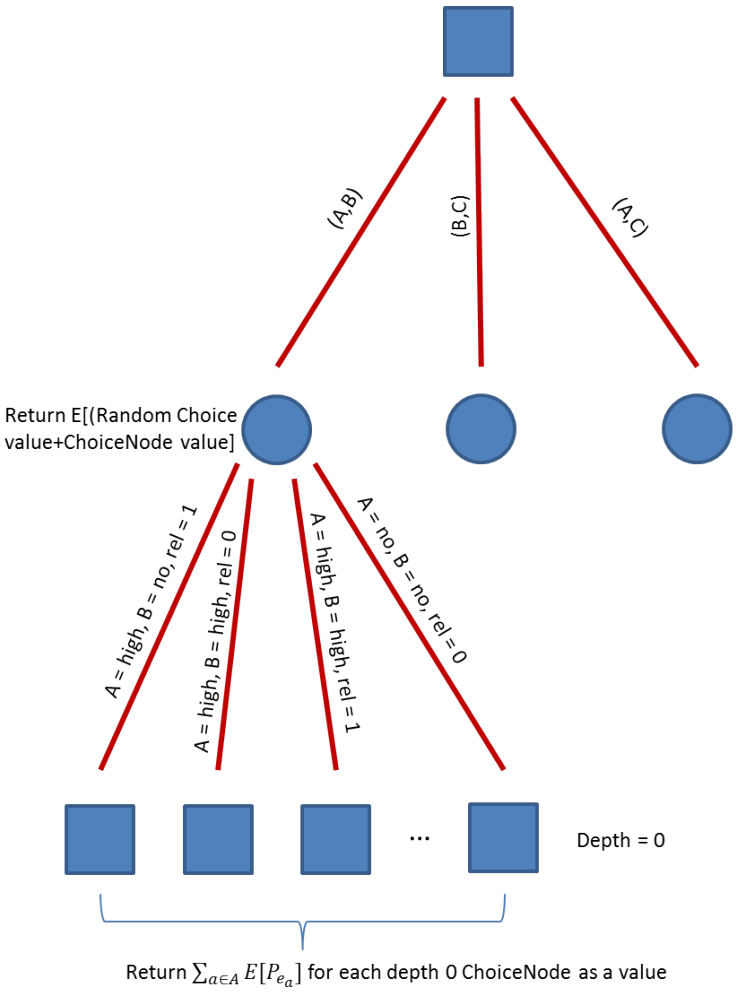


Figure 4.1: A partial diagram of `ChoiceNode` (square) and `RandomNode` (circle) objects created during a single iteration of the FHM algorithm with depth = 1. `FHM()` begins by creating a `ChoiceNode` object and callings its `getVal()` method, denoted by the box at the top of the figure. Since the depth = 1, we create a `RandomNode` object for each possible choice and call their `getVal()` methods. The `getVal()` method for the `RandomNode` created by the (A,B) choice is shown. There are 18 possible values that can result from choosing (A,B), and Figure 4.1 enumerates four of them. For each of these values, a new `ChoiceNode` object is created with depth = 0. The `getVal()` method of each `ChoiceNode` returns an estimated value since depth = 0. The `RandomNode` then returns its value as an expectation over all values of the depth = 0 *ChoiceNodes*. This process is also completed, but not shown, for the (B,C) and (A,C) choices. The value of the top `ChoiceNode` is then calculated as the max value from the set of children `RandomNodes`,{(A,B), (B,C), (A,C)}. The edge associated with this value is selected as the next edge for screening.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 5:
Analysis

Chapter section summary:

5.1: Describes the computational performance of `GraphBuilder` as a function of graph size.
5.2: Parameter testing on the FHM algorithm and its effect on computational performance.
5.3: Preliminary analysis on six sub-graphs created with `MapBuilder`.
5.4: A comparison of `GraphBuilder` to an approach which doesn't account for dependence.
5.5: Algorithm performance as a function of the probability of sudden revelation.
5.6: Analysis of the effect of graph structure on model and algorithm performance.
5.7: Algorithm performance when the knowledge gained from repeated screening of relevant sources diminishes.

5.1 Software Performance

The average iteration time of an algorithm increases exponentially as we increase the number of D_u levels, however varying the number of P_e levels has little effect on average iteration time.

We conduct some exploratory analysis on the performance of the `GraphBuilder` software to determine how varying attributes of the model affect the computational tractability.

We start our testing with a graph of 458 nodes and 490 edges created with the infection - targeted sub-graph creation method. Our objective is to measure the average iteration time of Softmax with different numbers of D_u and P_e levels. The probability of sudden revelation, c, is set to zero to ensure that factor sizes remain constant. We define the average iteration time as the amount of time in seconds it takes to select an item, screen it, and perform any subsequent inference calculations. The number of D_u levels is varied from three to five, and the number of P_e levels from two to five, and the results shown in Figure 5.1.

The average iteration time appears to increase exponentially as we increase the number of D_u levels. Varying the P_e levels has almost no effect on average iteration time. While additional P_e levels do require more computation, the effects appear to be overshadowed by other operations. These iteration times represent a worst case for algorithm performance, as any sudden revelations will result in smaller factors and faster inference calculations.

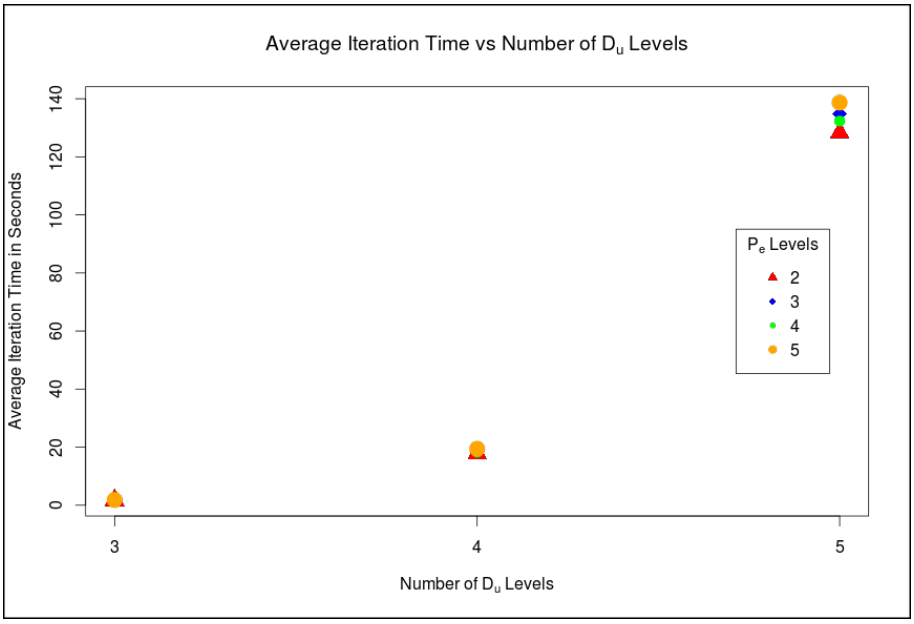


Figure 5.1: A plot of the average iteration time for Softmax on a graph of 458 nodes and 490 edges created with the infection - targeted method. The number of D_u levels is varied from three to five, and the number of P_e levels from two to five. The average iteration time appears to increase exponentially as we increase the number of D_u levels, while varying the number of P_e levels has almost no effect.

Next, we fix the number of D_u levels to three, the number of P_e levels to two, and run Softmax on graphs of increasing size. Graphs ranging in size from 400 edges to 2,500 edges are created with the infection - targeted method, and the results plotted in Figure 5.2. The average iteration time appears to be approximately linear in the number of edges in the graph.

5.2 FHM Performance

FHM algorithm performance remains strong even when the algorithm is extremely limited in the number of choices it can consider before selecting an item.

In Section 4.8 we identified that even at depth one, the computational tractability of FHM might be poor if each *ChoiceNode* object must consider selection of the next item to screen from all available edges in the network. We implement a user provided restriction to limit these choices while still providing the opportunity for exploration, and conduct parameter testing on the FHM algorithm to access if the computational tractibility can be improved without damaging its performance. We test the performance of the unrestrained (full) and choice limited FHM against the perfect selection method and Pure Exploitation, with results shown in Figure 5.3.

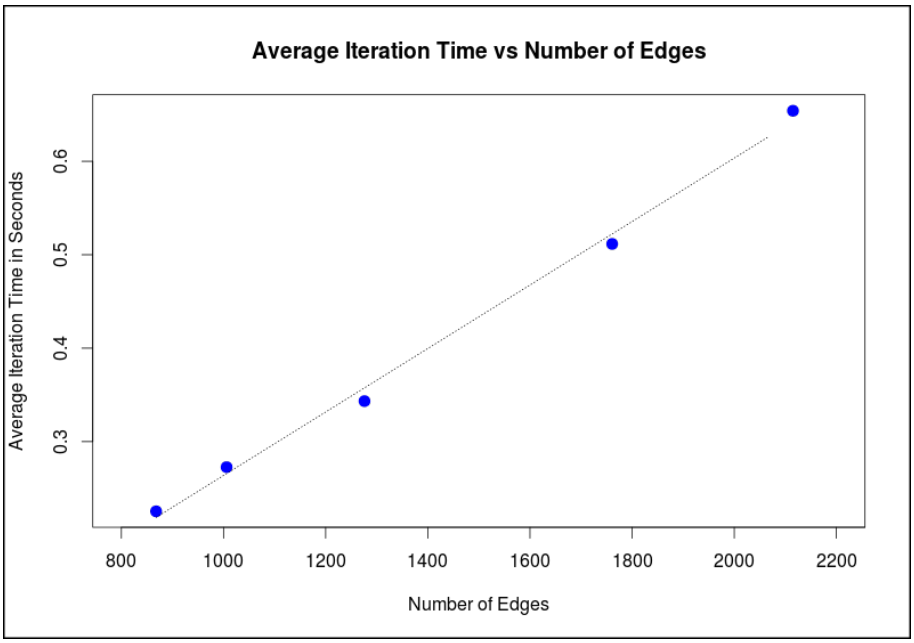


Figure 5.2: A plot of the average iteration time for Softmax on graphs of increasing size, showing that the average iteration time is approximately linear in the number of edges in the graph. All graphs were created with the infection - targeted method. The number of D_u levels is fixed at three, and the number of P_e levels is fixed at two.

For our test network, we choose the expanded Tanzanian terrorist network used by Nevo (2011, Chap V, pg 82). The network consists of 17 relevant nodes and 17 irrelevant nodes, with 49 edges, and is shown in Figure 5.4. We record the number of relevant conversations identified over 20 runs of 300 iterations each, and plot the results using a *beanplot* (Kampstra, 2008). The gray horizontal lines denote the observed number of relevant conversations identified in each run, while the black line extending from each plot represents the mean. The shape of the bean represents the shape of the distribution.

Both the full and choice limited FHM appear to have a slightly smaller variance than Pure Exploitation, with no discernible performance loss evident between the full and choice limited versions. Analysis of individual algorithm traces, shows that even when severely choice limited, the amount of exploration performed in early iterations is fairly consistent. Exploration happens when the algorithm selects an edge from among the possible choices that do not have the highest $E[P_e]$ values. In the choice limited algorithms, these are the edges that are selected randomly to be possible choices. This early exploration allows FHM to more quickly identify the high p_e valued edges to exploit in later iterations.

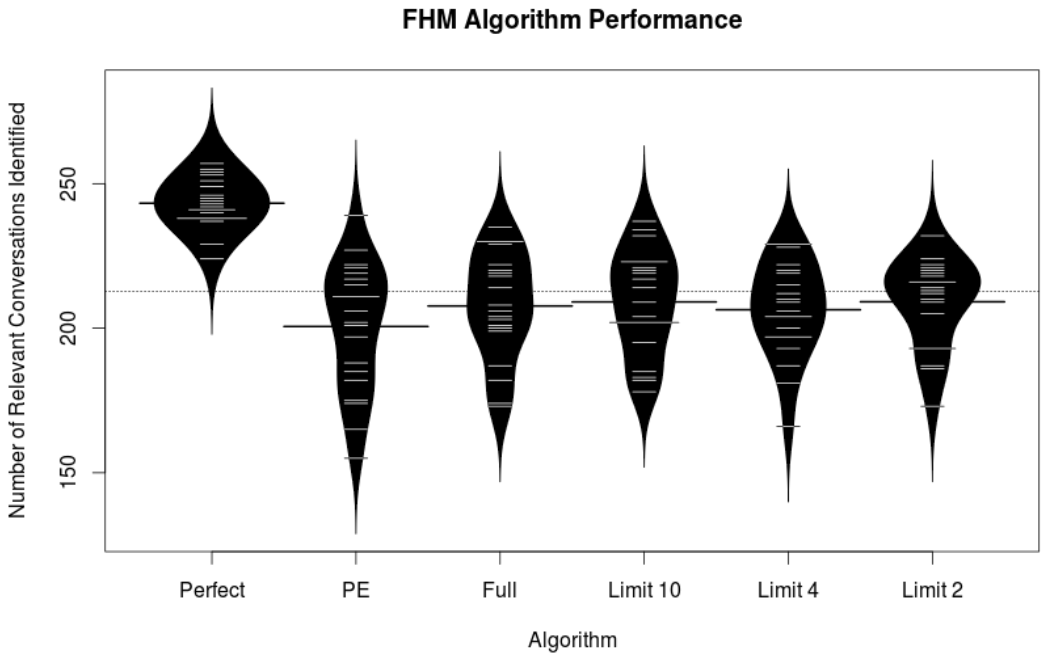


Figure 5.3: We test both the unrestrained (full) and choice limited FHM against the perfect selection method and Pure Exploitation. 20 runs of 300 iterations each are conducted on a graph of 34 nodes and 49 edges and we record the number of relevant conversations identified. Detailed network topography can be found in Nevo (2011, Chap V, p 82), and a visual in Figure 5.4. The results are displayed using a *beanplot*. The gray horizontal lines denote the observed number of relevant conversations identified for each run, while the black line extending from each plot represents the mean. The shape of the bean represents the shape of the distribution. Both the full and choice limited FHM appear to have a slightly smaller variance than Pure Exploitation, with no discernible performance loss evident between the full and choice limited versions.

5.3 Preliminary Algorithm Comparison

Algorithm performance is highly dependent on graph structure. Networks with a very low density of relevant items, where the relevant nodes do not cluster, have performance only slightly above the random selection method. FHM appears to be the most resilient to variation in structure.

In this section we perform some preliminary testing to determine how the algorithms perform when run on different graph structures.

5.3.1 Test Networks and Algorithm Parameters

We use the six example sub-graphs created in Chapter III for our initial algorithm testing. Summary statistics for the targeted and naive versions of the deep, wide, and infection graphs can

be found in Figures 3.5, 3.7, and 3.9 respectively. Each Algorithm is run 20 times with 300 iterations on each of the six graphs, and the number of relevant items screened is recorded. Initial parameters for the algorithms are taken from Nevo (2011) and are listed in Table 5.1.

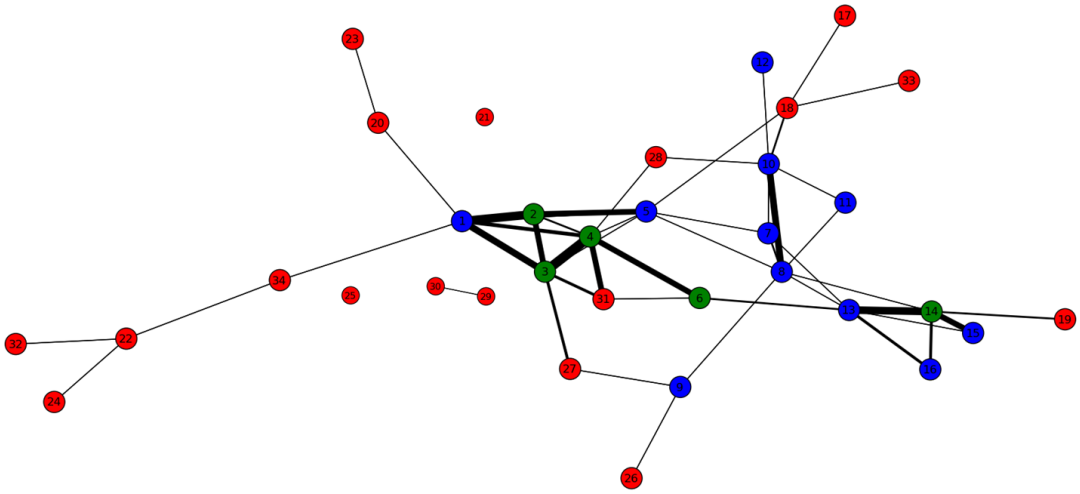


Figure 5.4: The expanded Tanzanian terrorist network. Five nodes have a high relevance value, 11 nodes are of medium relevance, with the rest having low relevance. Higher edge thicknesses denote high p_e values.

5.3.2 Results and Analysis

Figure 5.5 contains the results. Results are segregated by algorithm, with the average number of relevant items identified for each graph type shown. The random and perfect selection methods are included as worst and best case bounds for performance.

Table 5.1: Chosen parameters for initial algorithm performance comparisons. The FHM Choice Limit edges are picked using the method described in Section 4.8.

Algorithm	*Parameter*	*Value*
Random	None	
Perfect	None	
Pure Exploitation	None	
Softmax	Temperature	.08
VDBE	δ	.1
	σ	.4
	Temperature	.25
FHM	Depth Limit	1
	Choice Limit	10

The error bars denote a 95 percent confidence interval for the average number of relevant items identified, calculated using a t-distribution. All algorithms run on the deep and wide graphs created using the naive sub-graph creation method performed very poorly. From Figure 3.5 and 3.7 we can see that the number of relevant items available for screening was extremely low compared to the total number of available items. An analysis of the distribution of edge p_e values shows very little variation with most p_e values being extremely low. With the relevant items therefore contained on only a few select edges, and with these edges surrounded by low p_e edges, the algorithms had a difficult time identifying the optimal edges to screen. Additionally, these graphs do not contain clusters of relevant nodes. This breaks the assumption of the inference model that dependence between the nodes exists, and leads to poor performance.

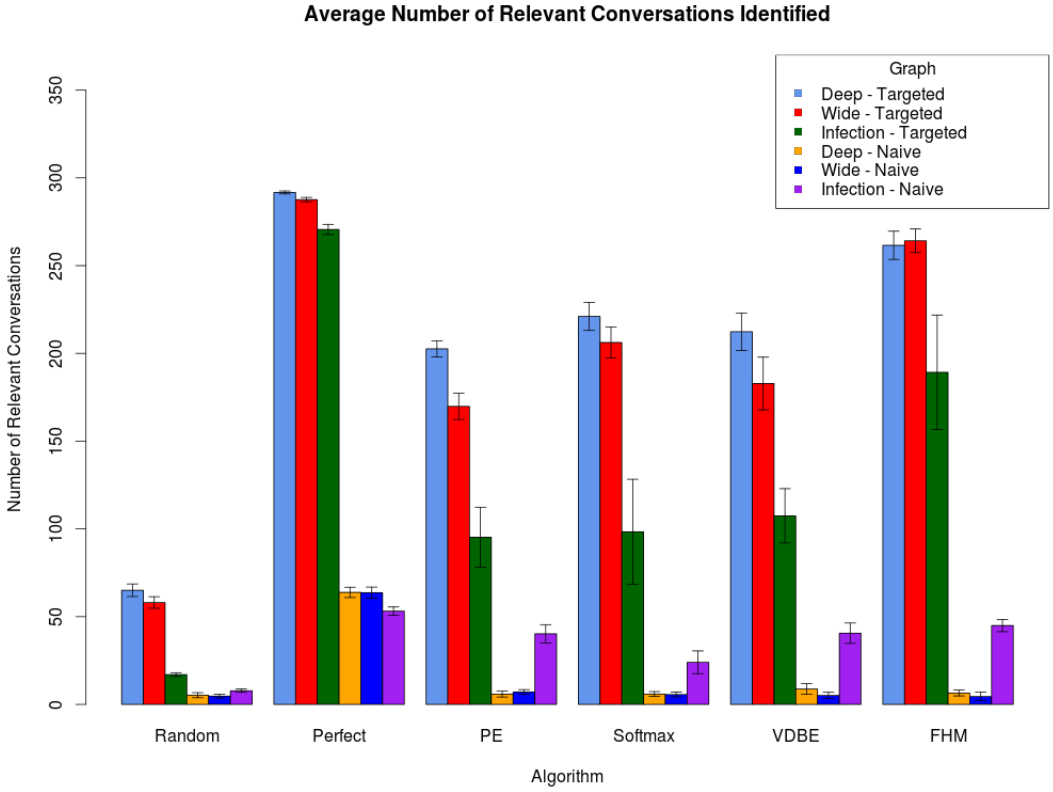


Figure 5.5: Results of algorithm testing on the six sub-graphs created in Chapter III. Results are segregated by algorithm, with the average number of relevant items identified for each graph type. The error bars denote a 95 percent confidence interval calculated using a t-distribution. The random and perfect selection methods are included as worst and best case bounds for performance. Algorithms run on the deep and wide graphs created using the naive sub-graph creation method performed very poorly, while FHM appears to demonstrate robustness across the deep - targeted, wide - targeted, infection - targeted, and infection - naive sub-graph construction techniques.

The performance of FHM appears to be fairly high on the deep - targeted, wide - targeted, infection - targeted, and infection - Naive sub-graph construction techniques, suggesting that FHM performance might be more robust on different graph structures than the other algorithms. In the deep - targeted, wide - targeted, and infection - targeted graphs FHM clearly outperforms Pure Exploitation, Softmax, and VDBE. The performance of Pure Exploitation, Softmax, and VDBE appears to be fairly similar across the six tested graphs, with Softmax generally having a slightly higher average number of relevant conversations identified. Since VDBE performance does not appear to be notably greater than Softmax and FHM, and the algorithm requires three user supplied tuning parameters, we disregard it for further trials.

5.4 The Value of the Knowledge Model

On graphs where relevant items are clustered together, `GraphBuilder`, which models dependence between p_e values, consistently outperforms the naive approach of `GraphBuilderNaive` across all tested algorithms. In the cases where p_e values are not highly correlated, FHM provides the best performance.

With the results of Section 5.3.2 showing that graph structure impacts the performance of the `GraphBuilder` software, we conduct additional testing to attempt to understand the topology under which the model performs well. The proposed advantage of the model implemented in the `GraphBuilder` software is that it is able to account for likely correlation in edge p_e values, which we consider a realistic attribute of real-world intercepted intelligence networks. That is, when the model screens either a relevant or irrelevant item on a particular edge, it updates not only the $E[P_e]$ value of the screened edge, but also edges elsewhere in the graph structure. A natural comparison, therefore, is to test this model against one that implements a more naive approach, that is a model that considers the $E[P_e]$ values as independent, updating only the $E[P_e]$ value of the screened item's edge.

We implement a naive version of the `GraphBuilder` software in a new module, `GraphBuilderNaive`, with full API documentation provided in Appendix D. `GraphBuilderNaive` constructs a separate graphical model for each edge in an intercepted intelligence network, using the same construction technique as `GraphBuilder`. When an item is screened, the graphical model corresponding to only that edge is updated, leaving the $E[P_e]$ values throughout the rest of the graph unchanged.

We test the performance of `GraphBuilder` against `GraphBuilderNaive` on two different graphs.

The first is the deep - targeted Enron sub-graph shown in Figure 3.4. The second is the Tanzanian terrorist network, shown in Figure 5.4. We compare the performance of Pure Exploitation, Softmax and FHM over 20 runs of 300 iterations each, with the results displayed in Figure 5.6.

On the Tanzanian terrorist network, we can see that the `GraphBuilder` software outperforms its naive counterpart on all three algorithms. We calculate a $100(\alpha - 1)$ confidence interval for the percent change in the average number of relevant items identified, $\%chg$, as

$$\%chg \pm t_{\alpha/2,n-1} * SE_{\%chg} \tag{5.1}$$

where

$$\%chg = \frac{R_k - R_n}{R_n} * 100 \tag{5.2}$$

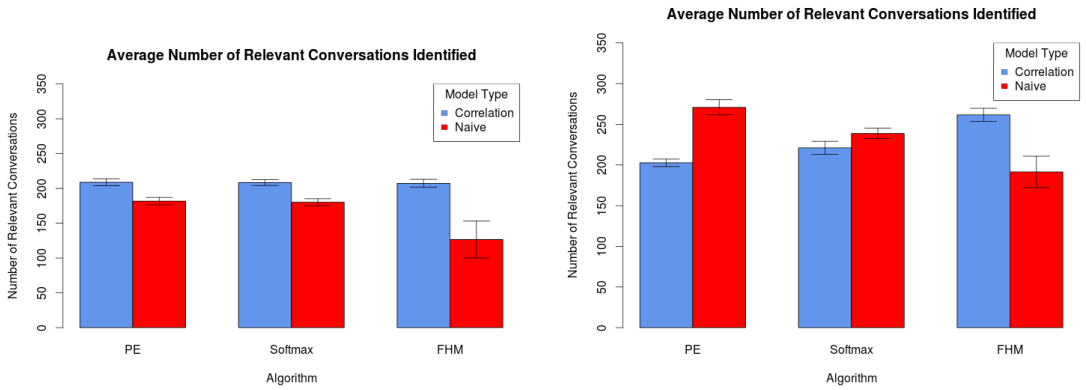
and R represents the average number of relevant items identified in the `GraphBuilder` (k) and `GraphBuilderNaive` (n) runs. The standard error, $SE_{\%chg}$, is calculated as

$$\left|\frac{R_k}{R_n}\right| * \sqrt{\frac{SE_k^2}{R_k^2} + \frac{SE_n^2}{R_n^2}} * 100 \tag{5.3}$$

At a 95 percent confidence level, for Pure Exploitation, there is a $14.7 \pm 5.5\%$ improvement, for Softmax a $15.6 \pm 5.2\%$ improvement, and for FHM, a $63.8 \pm 13.9\%$ improvement. We note that while Pure Exploitation and Softmax appear to have reasonable performance in the naive model, FHM performs very poorly.

When run on the deep - targeted sub-graph, `GraphBuilderNaive` outperforms `GraphBuilder` on two of the three algorithms. For Pure Exploitation, there is a $25.2 \pm 3.1\%$ decrease, and for Softmax the decrease is $7.4 \pm 4.2\%$. On FHM `GraphBuilder` outperforms `GraphBuilderNaive` by $36.6 \pm 7.3\%$.

Analysis of the deep - targeted sub-graph provides insight on the poor performance of `GraphBuilder`. With a largest maximal clique of size three, the graph doesn't contain any clusters of like relevance valued nodes. Edges with high p_e values are adjacent to edges with low p_e values, and no clear correlation of p_e values is evident.



(a) Tanzanian terrorist network

(b) Deep - targeted subgraph

Figure 5.6: A comparison of the performance between `GraphBuilder`, which models dependence between p_e values, and `GraphBuilderNaive`, which only updates $E[P_e]$ values for the edge that is screened. The error bars denote a 95 percent confidence interval for the average number of relevant items identified, calculated using a t-distribution. When the graph contains cliques of relevant items, such as in the Tanzanian terrorist network, the `GraphBuilder` model consistently outperforms its naive counterpart. When high p_e edges are obscured by adjacent low p_e edges, `GraphBuilder` can perform worse than the naive version, although FHM performance remains fairly robust.

This makes `GraphBuilder` perform poorly, for if an algorithm finds a relevant item on a particular edge, it will raise the $E[P_e]$ values on the adjacent edges, even though on this particular graph they are irrelevant. In contrast the Tanzanian terrorist network contains a maximal clique of five high and medium relevance nodes, along with several smaller like relevance valued cliques, so the updated $E[P_e]$ values calculated by `GraphBuilder` are more likely to be correct. In summary, if the graph does not bear out the dependence assumptions in the model, the model will likely perform poorly because it will direct screening in the wrong place.

Softmax and in particular, FHM, appear to perform better than Pure Exploitation in graph structures that do not contain clusters of like relevance nodes. An analysis of some algorithm traces shows that these algorithms are more likely to explore in the early iterations, and by doing so can identify a high p_e edge to exploit. In contrast, Pure Exploitation contains no exploration mechanism and therefore in a structure where the high p_e edges are not generally adjacent, as in the case of the deep - targeted sub-graph, it performs poorly.

5.5 Sudden Revelation

On graphs where relevant nodes are not clustered together, the probability of sudden revelation can markedly increase algorithm performance. Additionally, algorithms with a propensity for exploration early in the iteration cycle are more robust.

We continue our analysis by testing the results of varying the probability of sudden revelation. We perform this analysis on the two graphs used in Section 5.4, the Tazanian terrorist network and the deep - targeted sub-graph. The probability of sudden revelation is varied from zero to .1, with 20 runs of 300 iterations for Pure Exploitation, Softmax, and FHM. A `GraphBuilderNaive` run with a sudden revelation probability of .1 is also provided for comparison purposes. Results are provided in Figures 5.7 and 5.8.

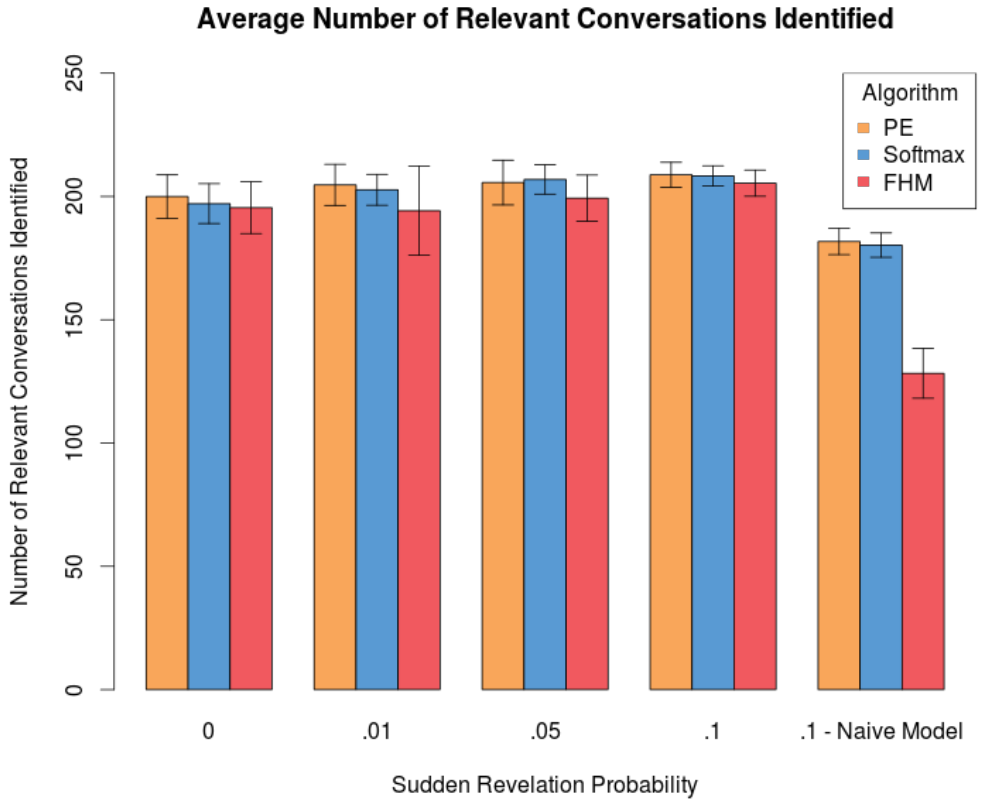


Figure 5.7: Algorithm performance under varying probabilities of sudden revelation using the Tanzanian terrorist network. The error bars denote a 95 percent confidence interval for the average number of relevant items identified, calculated using a t-distribution. All three algorithms show nearly identical performance across sudden revelation probabilities ranging from 0 to .1. A comparison to `GraphBuilderNaive` is provided for comparison.

Figure 5.7 shows the performance of Pure Exploitation, Softmax, and FHM when run on the Tanzanian terrorist network. All three algorithms show nearly identical performance across the entire range of sudden revelation probabilities. A comparison to `GraphBuilderNaive` shows that regardless of the sudden revelation probability, all the algorithms outperform the naive approach.

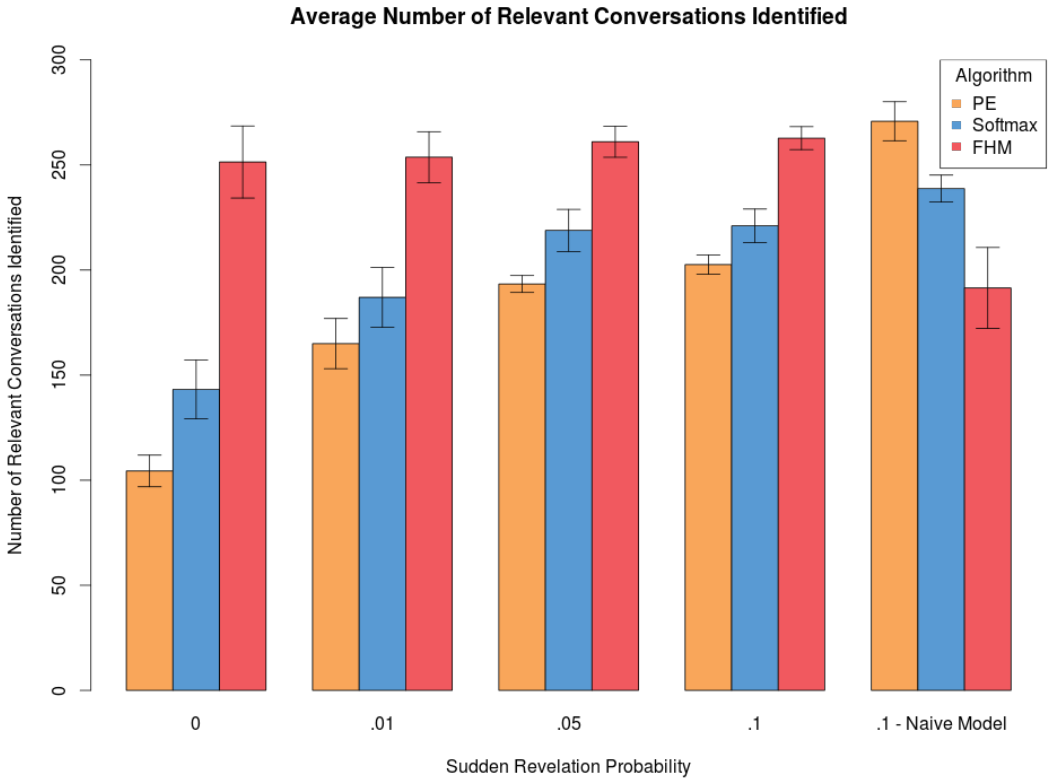


Figure 5.8: Algorithm performance under varying probabilities of sudden revelation using the deep - targeted sub-graph. The error bars denote a 95 percent confidence interval for the average number of relevant items identified, calculated using a t-distribution. Increasing the probability of sudden revelation notably improves the performance of all three algorithms, although `GraphBuilderNaive` continues to outperform `GraphBuilder`. FHM shows remarkable resilience, with performance almost equaling that of `GraphBuilderNaive` Pure Exploitation. Analysis shows that the propensity of FHM to explore in the early iterations allows it to find and exploit high p_e edges earlier.

Figure 5.8 shows the performance of Pure Exploitation, Softmax, and FHM when run on the deep - targeted sub-graph. On this graph, increasing the probability of sudden revelation from zero to .1 notably improves the performance of all three algorithms. Pure Exploitation shows a $94.1 \pm 14.7\%$ improvement, Softmax a $54.4 \pm 16.1\%$ improvement, and FHM, a $4.5 \pm 3.5\%$

improvement, with performance increasing as a function of the sudden revelation probability. As shown in Section 5.4, the performance of `GraphBuilderNaive` is better for Pure Exploitation and Softmax. The FHM algorithm performs astonishingly well when compared to Softmax and Pure Exploitation. An analysis of algorithm traces shows that FHM's propensity to explore early in the iteration cycle allows it to find a high p_e edge much earlier than other algorithms, and it can then exploit this edge for the remaining iterations. It appears that on graph structures that do not contain like relevance valued nodes in clusters, algorithms that allow for more exploration in early iterations are far more likely to find high p_e edges than algorithms that do not explore. Because the value of a relevant item remains constant, once the algorithm finds a high value edge, it can exploit it for the remainder of the available time.

5.6 Clustering

`GraphBuilder` outperforms `GraphBuilderNaive` by the largest margin in graphs where the density of high relevance nodes is neither very high nor very low.

With the analysis of the above sections showing that graph structure clearly impacts the performance difference between the `GraphBuilder` and `GraphBuilderNaive` models, we explore under which types of structures `GraphBuilder` has the greatest advantage.

We conduct our testing on four graphs. Each has two node relevance levels, low and high. High relevance items are located together in maximal cliques of size four. True p_e values between high relevance valued nodes are .9, while all other edges have a p_e value equal to .1. Each graph contains a different number of high relevance maximal cliques, ranging from one to four. Graph topography is shown in Figure 5.9.

We test the performance of Pure Exploitation, Softmax, and FHM on the four graphs in Figure 5.9, using both `GraphBuilder` and `GraphBuilderNaive`, conducting 20 runs of 300 iterations for each combination of model, algorithm, and graph. The sudden revelation probability is fixed to .1 for all examples, and the results are shown in Figure 5.10.

From Figure 5.10, we can see that when the density of relevant nodes is very low, as in the case of the one cluster graph, that although `GraphBuilder` outperforms `GraphBuilderNaive` for Softmax and FHM, the performance difference is quite minimal. The results for the four cluster graph, where the density of relevant items is very high, is similar, with `GraphBuilder` achieving a noticeable but not distinct performance advantage over `GraphBuilderNaive`.

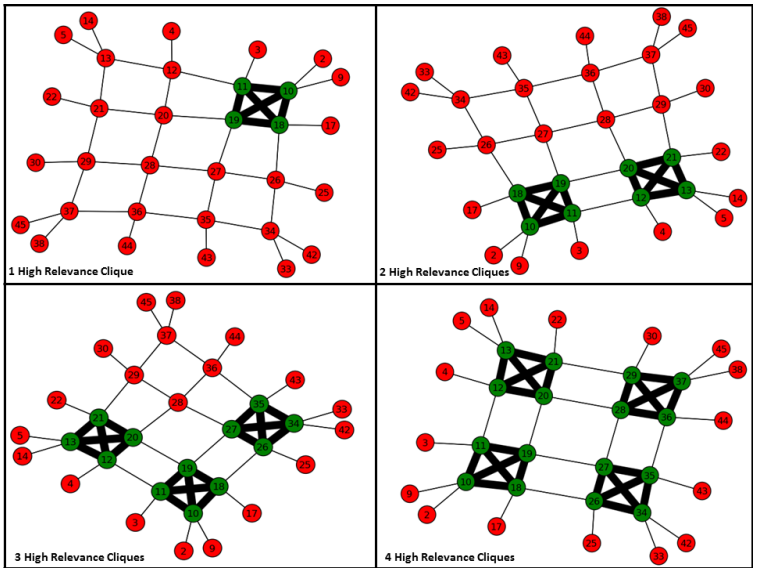


Figure 5.9: Four artificially constructed graphs designed to test the affect of graph structure on algorithm performance. Green nodes have high relevance, with the thick edges between high relevance nodes having $p_e = .9$. Red nodes have a low relevance value with all adjacent edges having $p_e = .1$. Each graph contains a different number of size four maximal cliques of high relevance value nodes.

In the graphs with medium density of high relevance items, namely the two and three cluster graphs, `GraphBuilder` outperforms `GraphBuilderNaive` by a much larger margin. Although these graphs are idealized structures, they suggest that if an intercepted intelligence network contains pockets of relevant nodes surrounded by lower relevance noise, that a correlation based approach is likely to outperform a naive one. In the two cluster graph, algorithms that contain more exploration, such as Softmax and FHM, outperform Pure Exploitation, as they're more likely to uncover the second maximal clique of high relevance nodes.

5.7 Knowledge Value Reduction

`GraphBuilder` performs quite well, even when the value of subsequent relevant items from an already exploited edge decreases.

In previous sections, we assume that the value of a relevant item on a particular edge is either one or zero, and use a metric of average number of relevant conversations identified to compare the performance of different models and algorithms. It's probable however, that in real world intelligence networks, the value of a relevant piece of information is not always the same. We envision a scenario where the value of the first relevant item identified on an edge is higher than subsequent relevant items, due to information being repeated in the subsequent items.

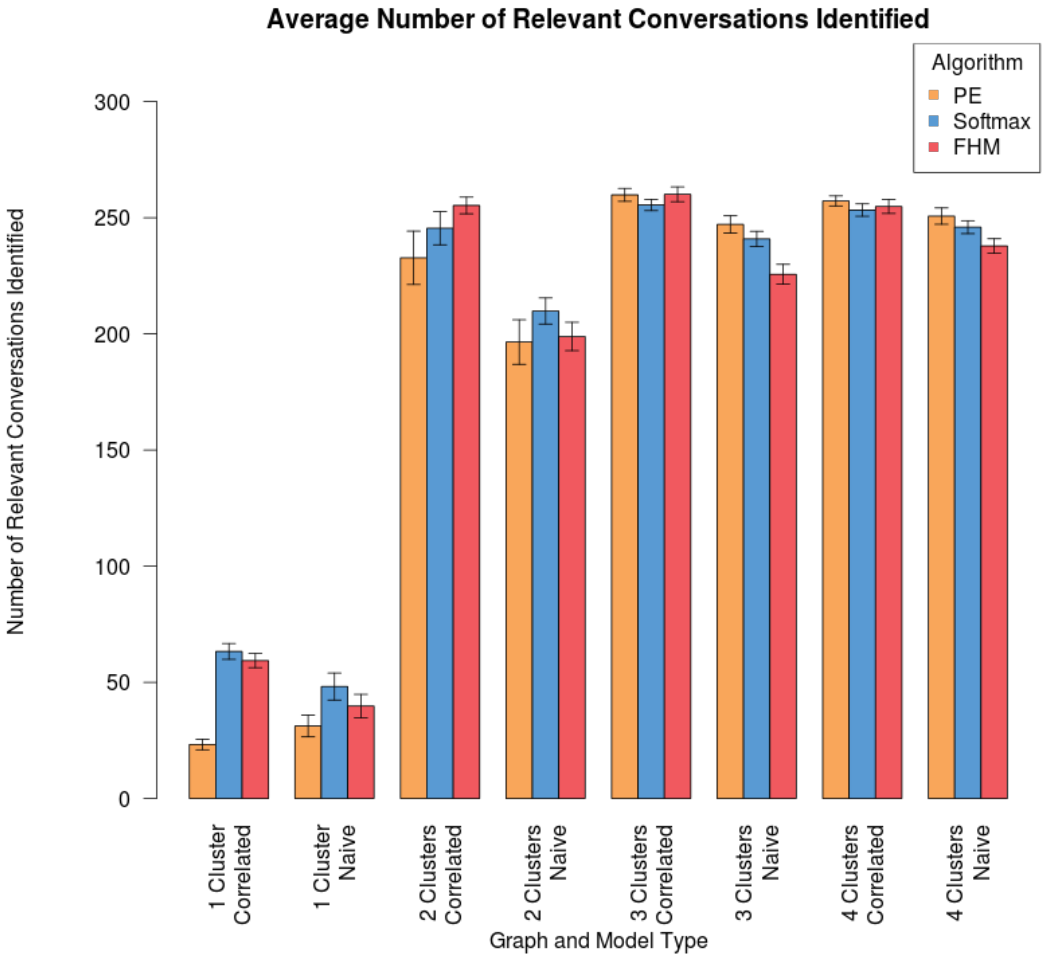


Figure 5.10: Algorithm performance results when both `GraphBuilder` and `GraphBuilderNaive` are run on the graphs shown in Figure 5.9. The error bars denote a 95 percent confidence interval for the average number of relevant items identified, calculated using a t-distribution. The density of relevant items within the graph appears to have a large impact on performance between the correlation and naive approaches. When the density is very low or very high, as in the 1 and 4 cluster graphs, the performance difference between `GraphBuilder` and `GraphBuilderNaive` is very minimal. In graphs of medium density, such as the 2 and 3 cluster graphs, `GraphBuilder` notably outperforms `GraphBuilderNaive`.

As described in Section 4.2, the `Algorithms` module is capable of accepting a user supplied knowledge reduction function. We therefore implement a function where the value of a relevant item discovered on an edge decreases exponentially with each additional relevant item discovered. For example, if the processor screens an item on an edge that has not been explored, and finds it to be relevant, it is assigned a value of 1. If the value of the exponential decrease is .1,

the next relevant item screened on that edge would have the value $(1-r)^K = (1-.1)^1 = .9$, where r is the rate, and K is the number of relevant items already screened on that edge.

We test varying rates of reduction from .025 to .2 on the Tanzanian terrorist network of Figure 5.4, with 20 runs of 300 iterations each conducted for each algorithm. We sum the value of the knowledge for each relevant item screened, with the results shown in Figure 5.11. We also include the perfect and random selection methods as upper and lower bounds.

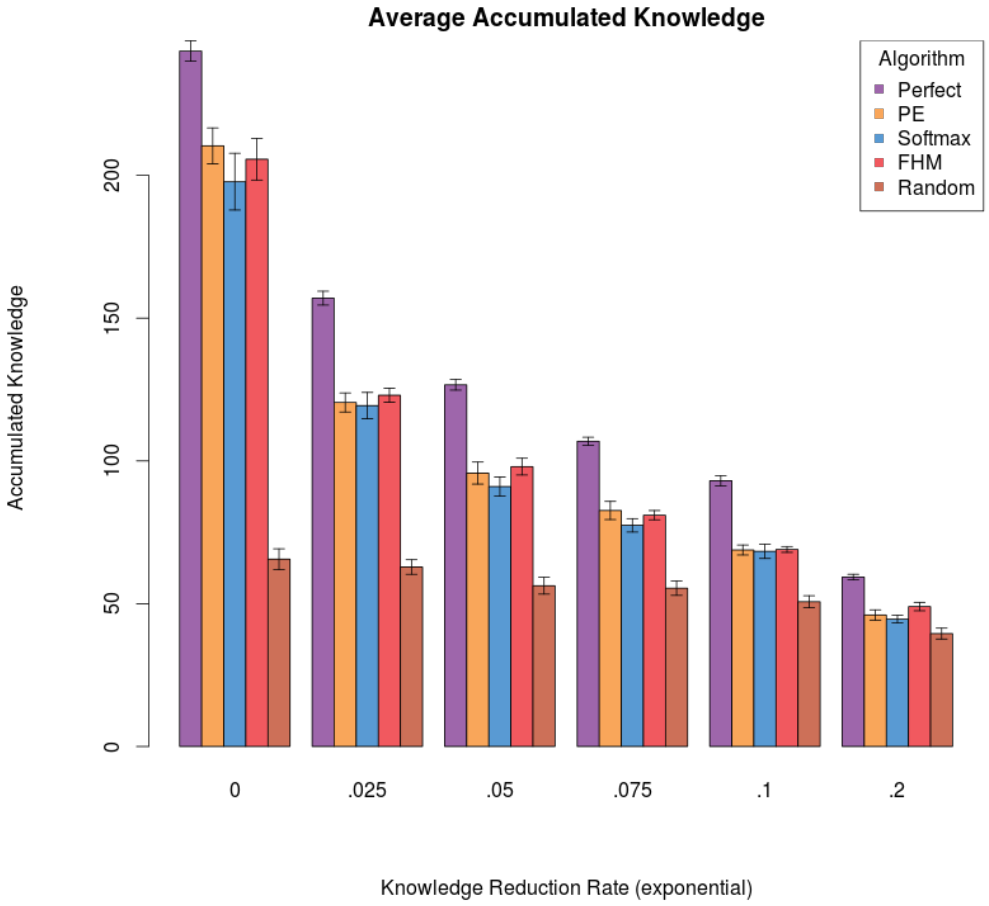


Figure 5.11: Algorithm performance results for the Tanzanian terrorist network. A reduction function is implemented, where the value of a relevant item discovered on an edge decreases exponentially with each additional relevant item discovered. The error bars denote a 95 percent confidence interval for the average knowledge accumulated, calculated using a t-distribution. `GraphBuilder` performs well, with all three algorithms (Pure Exploitation, Softmax, and FHM) outperforming the random selection method.

From Figure 5.11, we can see that `GraphBuilder` performs well even with the exponential decrease function applied, with all three algorithms (Pure Exploitation, Softmax, and FHM) outperforming the random selection method. For the baseline case with zero reduction, the

three algorithms achieve approximately 85 percent of the performance of the perfect selection method. For the five exponential knowledge reduction rates, the algorithms achieve a range of approximately 72 to 79 percent of the perfect selection method's performance, showing that performance loss from the optimal method is consistent over increasingly severe reduction rates. At rates greater than .2, the available knowledge degrades too fast to allow for proper algorithm performance.

CHAPTER 6:
Conclusion

In this chapter we summarize the results of our analysis, suggest some possible extensions to the mathematical model and software, and propose additional follow-on research.

6.1 Summary and Main Conclusions

In this thesis, we focus on the challenge of an intelligence processor faced with finding the maximum amount of relevant information in a potentially overwhelming volume of communications data.

From Nevo (2011), we describe a mathematical model of the intelligence screening process, which uses techniques from graphical models, social networks, random fields and Bayesian learning. Based on this model, we construct a library of software tools:

1. `GraphBuilder`: Uses the above mathematical model and methodology, and is capable of reading in a large graph representing an intercepted intelligence network and creating an object that represents the knowledge of the processor. Methods are supplied which allow for updating of the processor's knowledge as items are screened. The software is capable of quickly calculating the joint probability distribution for $\overline{D}$.

2. `GraphBuilderNaive`: Implements a naive version of the mathematical model, constructing a separate `GraphBuilder` object for each edge in the network. In this model, the knowledge a processor obtains from screening an item only affects the $E[P_e]$ value for the screened edge.

3. `MapBuilder`: Allows for the efficient generation of test networks representing intercepted intelligence networks from the Enron corpus. Methods for data visualization, statistics collection, network trimming, and IO are provided.

4. `Algorithms`: Contains heuristic algorithms for the screening optimization problem, as well as bounding selection methods representing best and worst case screening scenarios. Pure Exploitation, Softmax, Value-Difference-Based-Exploration (VDBE), Wide Exploration First (WEF), and Finite Horizon Markov Decision Process (FHM) algorithms are

implemented.

Using these software tools, we evaluate the run-time performance of `GraphBuilder`, establish parameters for the efficient running of FHM,compare `GraphBuilder` to `GraphBuilderNaive`, and evaluate the effect of varying model parameters and network structure. Detailed analysis is provided in Chapter V, with some insights provided in Sections 6.1.1 and 6.1.2.

6.1.1 Main Insights

1. If the graph does not bear out the dependence assumptions in the model, the model will likely perform poorly as it will direct screening in the wrong place. On graphs where relevant items are clustered together, `GraphBuilder`, which models dependence between p_e values, consistently outperforms the naive approach of `GraphBuilderNaive` across all tested algorithms. In the cases where p_e values are not highly correlated, FHM provides the best performance. This might be of concern to intelligence agencies if the methods of collection only obtain a small fraction of the entire communications network. Under such a scenario, the graph structure might not have dense enough clusters of relevant sources.

2. `GraphBuilder` outperforms `GraphBuilderNaive` by the largest margin in graphs where the density of high relevance nodes is neither very high nor very low. This suggests that if an intercepted intelligence network contains pockets of relevant nodes surrounded by lower relevance noise that a correlation based approach is likely to outperform a naive one.

3. On graphs where relevant nodes are not clustered together, the probability of sudden revelation can markedly increase algorithm performance. Additionally, algorithms with a propensity for exploration early in the iteration cycle, such as FHM, are more robust. This is because when the value of a relevant item remains constant, once the algorithm finds a high p_e valued edge, it can exploit it for the remainder of the available time.

6.1.2 Further Insights

1. `GraphBuilder` performs quite well even when the value of knowledge obtained from subsequent relevant items screened from an already exploited edge decreases. This condition might happen when information is repeated on subsequent relevant items that are

screened, lowering their value.

2. The average iteration time of an algorithm increases approximately exponentially as we increase the number of discrete node relevance (D_u) levels , however varying the number of edge relevance (P_e) levels has little effect on average iteration time. Total algorithm run time grows in approximately linear time with the number of edges in the graph.

3. FHM performance remains strong even at depth zero and with the algorithm extremely limited in the number of edge choices it is allowed to consider for selection.

6.2 Possible Extensions of the Model and Software

We propose several extensions to the model and software which could increase the realism and fidelity of future analysis and exploration.

Further FHM Modifcations. In section 4.8, we describe a heuristic to improve the computational tractability of FHM. While this choice limiting method drastically improves the performance of the algorithm at depth zero, the large number of *RandomNode* objects that must be created for each choice still results in unacceptably low performance at deeper depths.

To run FHM at depths greater than zero, sampling could be used to calculate the expectation at each `RandomNode`. For example, in Table 4.1, we enumerate the 18 possible outcomes of choosing an edge in a graph with two node relevance (D_u) levels. Rather than calculating the expectation over all 18 values, we could take the expectation over a smaller random sample of the outcomes. This would result in much faster run times and allow for testing of the algorithm at greater depth.

Extensions to Sudden Revelation. As described in Chapter II, the relevance of a node is either known or unknown. A node's relevance can only be discovered by screening an item on an edge to which it is adjacent. In `GraphBuilder`, we implement a fixed probability of sudden revelation (c), and model the probability of discovering the node relevance value of either of the two nodes adjacent to the screened items' edge as independent of each other.

We suggest some possible extensions to the sudden revelation portion of the model which would require only minor software changes:

1. Screening an item on an edge might reveal the relevance value of a non-adjacent node. `GraphBuilder` could be modified to account for a probability of discovering the relevance of a third party.

2. A conversation might include information which doesn't establish the relevance value of a node with certainty, but provides information that would make a particular relevance value more or less likely. This would require the model to update the probability distribution of the D_u's.

Time Constraints. We assume that the time to screen an item is fixed and identical for every item in the network. However, in a real-world problem, it's reasonable that items would require different amounts of time to screen. For example, a processor might take more time to screen a long communication than a short one. It's also possible that a processor is often able to quickly identify whether an item is relevant to the intelligence query, while in some cases, establishing the relevance could take considerable time. In cases where the processor is extremely time limited, this modification might require different screening strategies.

Processor Errors. In our model, we do not account for errors committed by the processor. These errors might take two principle types:

1. The processor might mis-identify a screened items' relevance.

2. The processor might mis-identify the relevance value of a node.

Expansion of MapBuilder. `MapBuilder` is capable of constructing test intercepted intelligence networks from the Enron corpus, but the module could be expanded to read in any arbitrary network. This would allow the data visualization, statistics collection, network trimming and IO functions to be utilized on a wider variety of structures.

Advanced Analysis Visualization Tools. Analysis of algorithm results is complicated by the high number of iterations and the computational complexity of the mathematical model. Software that allows for easier analysis of test results could prove helpful in understanding the run-time behavior of the algorithms. For example, a visualization tool that shows the changes in $E[P_e]$ values on the graph as the algorithm progresses could prove helpful in understanding why the algorithm chooses which edges to screen.

6.3 Future Research

In this section we suggest some additional future research topics.

Further Parameter Tuning and Topology Studies. In this thesis, we explore how changing model parameters, such as the probability of sudden revelation (c), impact the performance of the screening algorithms, however, the large number of possible model and algorithm parameter combinations means that more research should be done.

Additionally, we conduct some basic testing on the effect of graph topology on `GraphBuilder` and `GraphBuilderNaive`. Additional testing should be conducted to determine with more precision the conditions under which the models perform best.

Additional Algorithms. Our research focuses on testing four algorithms, Pure Exploitation, Softmax, VDBE, and FHM. Future research could be concentrated on identifying or developing additional heuristic algorithms to handle the information selection problem.

Techniques for Larger Graphs. Updating the probability distribution in `GraphBuilder` for $\overline{D}$ on graphs of up to several thousand edges can be computed in less than a second, however, this is still prohibitive for algorithms that require several inference calculations per iteration, such as FHM. The rate of change of $E[P_e]$ values decreases as the distance from the edge of the screened item increases. Larger graphs might be able to be processed more effectively with minimal loss of knowledge if the $\overline{D}$ probability distribution updates are done on smaller sub-graphs within the larger network, rather than on the entire structure. This would require significant software changes to `GraphBuilder`.

Real-World Data. The intercepted intelligence network parameters we utilize for our testing are not based on real-world data. Testing of the model on real-world intelligence data might be useful to further improve the model and validate its performance.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A:
GraphBuilder

A.1 Module GraphBuilderClass

Creates a Graphical Model Object representing the knowledge of an intelligence processor that can be used to test intelligence collection algorithms. Uses the gPy module developed by James Cussens at University of York, UK. Support documentation and further information concerning gPy can be found at: http://www-users.cs.york.ac.uk/jc/teaching/agm/

NetworkX graph structures suitable as intercepted intelligence networks for a graphical model can be constructed using the accompanying MapBuilder.py module.

A.1.1 Class GraphBuilder

The GraphBuilder class supports the creation of a graphical model and accompanying support functions required to test intelligence collection algorithms. Specific algorithms can be found in the Algorithms.py module.

Methods

__init__(*self*, *G*, *joint_prob_prefix*='joint', *pij_dij_file*='pij_dij.csv', *sij_file*='sij.csv', *c*=0.5, *precision*=5)

Construct a graphical model by reading in a NetworkX graph and accompanying probability distributions.

Parameters

G:	Graph to construct graphical model from. *(type=NetworkX Graph)*
joint_prob_prefix:	Prefix of file names that contain the joint distribution of the D_i's. *(type=int)*

`pij_dij_file:`	Filename for conditional probability distribution of P_ij, given, D_i, D_j. *(type=str)*
`sij_file:`	Filename for probability of P_ij, given S_ij. *(type=str)*
`precision:`	Number of digits to display in conditional probability tables. *(type=int)*

Return Value

Graphical model object.

---

count_factors(*self*)

Counts the number of factors in the graphical model.

Return Value

Number of factors in the graphical model.
(type=int)

---

count_remaining(*self*)

Calculates the remaining items available for screening in the model.

Return Value

The number of items available for screening.
(type=int)

---

edge_update(*self*, *edge*, *value*, *sumout*=`True`)

Update the graphical model after screening an item.

Parameters

`edge:` Edge to update.
(type=tuple)

`value:` Value of edge update.
(type=int)

`sumout:` If True, sum out S_ij factor after update.
(type=bool)

---

expected_di(*self*, *node*)

Displays the marginal probability distribution for a node.

Parameters

`node:` Graph node.
(type=str)

Return Value

Dictionary of probabilities.
(type=dict)

---

expected_pij(*self*, *edge*, *limit*=`'null'`, *args*=`[]`)

Calculates the expected P_ij for a requested edge.

Parameters

`edge`: Graph edge.
(type=tuple)

`limit`: Name of knowledge limiting function, if specified.
(type=str)

`args`: A list of knowledge limit function arguments.
(type=list)

Return Value

The expected P_ij for the requested edge.
(type=float)

---

fCalibrate(*self*)

Perform final calibration so that all factors associated with both cliques and separators are the appropriate marginal distributions. Makes permanent changes to the model. No further updates can be performed after calibration.

---

highest_expected_pij(*self*, *numEdges*=`None`, *limit*=`'null'`, *args*=`[]`)

Generates a list of edges sorted from highest to lowest expected probability for a relevant item.

Parameters

`numEdges`: Length of list to return.
(type=int)

`limit`: Name of knowledge limiting function, if specified.
(type=str)

`args:` A list of knowledge limit function arguments.
(type=list)

Return Value

Descending list of expected P_ij values in tuple form (Edge, Expected P_ij).
(type=list)

---

node_update(*self*, *node*, *value*)

Update a node relevance value from sudden revelation.

Parameters

`node:` Node to update.
(type=str)

`value:` Value of revelation.
(type=str)

---

normalise_factors(*self*)

Writes back the GFR with normalised factors from the JFR then creates a new JFR Note: not used in the current implementation.

---

print_GFR(*self*)

Writes the GFR structure to the screen with normalised factors.

---

print_JFR(*self*)

Writes the JFR structure to the screen.

---

print_factor(*self*, *factor*, *normalised*=`True`)

Display a factor from the model.

Parameters

`factor`: Factor to display, eg: ('H','I',('I','H')).
(type=tuple)

`normalised`: If True, normalise the factor values as a probability distribution.
(type=bool)

---

random_draw(*self*, *edge*)

Computes a random draw on an edge using the true p_ij value and returns the relevance value.

Parameters

`edge`: Edge on which to perform a random item draw.
(type=tuple)

Return Value

Relevance value of the item.
(type=int)

---

sij_add(*self*, *edread*, *edge*)

Add S_ij factor to the model for an edge update.

Parameters

`edread:` The number of items previously screened on the edge.
(type=int)

`edge:` Edge for which to add the S_ij factor.
(type=tuple)

Return Value

Name of the S_ij variable associated with the S_ij factor.
(type=str)

---

sudden_relevance_simple(*self*, *node*, *c*)

Computes the results of a sudden revelation realization on a node. Relevance is calculated with a fixed probability parameter.

Parameters

`node:` Node on which to perform a sudden revelation check.
(type=str)

`c:` Probability of sudden revelation for the node.
(type=float)

Return Value

(Boolean value for whether sudden revelation realization occurred, the node for which any sudden revelation occurred, and the value of the revelation).
(type=tuple)

---

sumout_sij(*self*, *sij*, *edge*)

Sum out an S_ij variable.

Parameters

`sij`: Variable to eliminate.
(type=str)

---

tCalibrate(*self*)

Performs a temporary calibration by performing a calibration on a copy of the model. Ensures that all factors associated with both cliques and separators are the appropriate marginal distributions. Used to calculate expected P_ij values without finalizing the model state.

---

true_pij_calc(*self*)

Calculates the true value of p_ij for every edge in the graph. Writes the results to the NetworkX Graph in self.

---

writeback_GFR(*self*)

Write back factor changes in the JFR to the GFR model with all factors normalised to prevent rounding error. Note: not used in the current implementation.

APPENDIX B:
MapBuilder

B.1 Module MapBuilder

A collection of functions for creating, manipulating, and displaying communication graphs constructed from the Enron corpus.

B.1.1 Functions

PEDist(*G*, *r*=`None`, *bins*=`None`, *writefile*=`False`, *datafile*=`'PEdata.csv'`)

Constructs a histogram of edge p_ij values in a graph.

Parameters

`G`: Graph.
(type=NetworkX Graph)

`r`: Lower and upper range of the histogram bins. If not provided, the range is [min,max] value.
(type=tuple)

`bins`: Enter an integer number of bins or a sequence giving the bins.
(type=int or list)

`writefile`: If True, write the p_ij data to a CSV file.
(type=boolean)

`datafile`: Name of output file.
(type=str)

Return Value

Distribution of edge p_ij values.
(type=histogram)

---

add_pij(*G*)

Calculates the true value of p_ij for every edge in the graph.

Parameters

`G`: Graph.
(type=NetworkX Graph)

Return Value

Graph.
(type=NetworkX Graph)

---

buildEnron(*outfile*=`'enron.sqlite3'`)

Constructs a SQLite3 database from the Enron corpus email database. Does not require re-running once the database is constructed.

Parameters

`outfile`: Name of the SQL database created.
(type=str)

---

buildGraph(*keys*=`['money', 'finance']`, *rels*=`['low', 'medium', 'high']`, *dbfile*=`'enron.sqlite3'`, *rebuild*=`True`)

Constructs a NetworkX Graph based on specified input parameters. The function interfaces with the Enron SQL database file constructed in the buildEnron function.

Parameters

`keys:` Keywords that denote relevant items.
(type=list)

`rels:` Node relevance values D_i.
(type=list)

`dbfile:` Filename of the Enron SQLite3 database constructed using buildEnron.
(type=str)

`rebuild:` If True, build new SQL tables in dbfile. This is only needed if the keys have changed since the last call.
(type=bool)

Return Value

Graph.
(type=NetworkX Graph)

---

conDist(*G*, *type*=`'total'`, *r*=`None`, *bins*=`None`)

Constructs a histogram of the number of conversations on the edges of a graph.

Parameters

`G:` Graph.
(type=NetworkX Graph)

`type:` 'total', or 'freq', determine which type of edge data to produce a distribution for. 'total' returns the distribution of all conversations. 'freq' returns the distribution of just relevant conversations.
(type=str)

`r:` Lower and upper range of the histogram bins. If not provided, the range is [min, max] value of specified type.
(type=tuple)

`bins`: Either an integer number of bins or a sequence giving the bins.
(type=int or list)

Return Value

Distribution for the number of conversations (relevant or total) on edges of the graph.
(type=histogram)

---

conv_Count(*G*)

Calculates the remaining number of available items for screening in the graph.

Parameters

`G`: Graph.
(type=NetworkX Graph)

Return Value

The number of available items left to screen.
(type=int)

---

create_di_csv(*G*, *rels*=`['low', 'medium', 'high']`, *prefix*=`'joint'`)

Creates the initial (prior) joint distributions for the D_i's. One distribution is created for every clique in the graph. Suitable for import into GraphBuilder.

Parameters

`G`: Graph.
(type=NetworkX Graph)

`rels`: Node relevance values for D_i.
(type=list)

`prefix`: Prefix for the filenames of the output files.
(type=str)

Return Value

Dictionary of joint probabilities.
(type=dict)

---

create_di_csv_naive(*G*, *rels*=`['low', 'medium', 'high']`, *prefix*=`'joint'`)

Creates the initial (prior) joint distribution for the D_i's. Naive approach that assumes all permutations of relevance values within a clique have equal probability. Used for comparison to the data driven approach in create_di_csv().

Parameters

`G`: Graph.
(type=NetworkX Graph)

`rels`: Node relevance values for D_i.
(type=list)

`prefix`: Prefix for filenames of output files.
(type=str @return dictionary of joint probabilities)

Return Value

Dictionary of joint probabilities.
(type=dict)

---

create_pij_dij_csv(*G*, *num_pijlevels*=2, *rels*=`['low', 'medium', 'high']`, *file*=`'pij_dij.csv'`)

Creates a conditional probability table for Pr(P_ij | D_i, D_j) using graph data. Suitable for import into GraphBuilder.

Parameters

`G:`	Graph. *(type=NetworkX Graph)*
`num_pijlevels:`	Number of discrete P_ij levels. *(type=int)*
`rels:`	Node relevance values for D_i. *(type=list)*
`file:`	Filename of output file. *(type=str)*

Return Value

Dictionary of conditional probabilities.
(type=dict)

---

drawGraphMaxNodes(*G*, *maxnodes*, *trim_freq*=0, *layout*='spring', *wf*=False)

Plots a graph to the screen. Colors nodes by their membership in the maxnode list. Is capable of saving the graph to a PDF file.

Parameters

`G:`	Graph. *(type=NetworkX Graph)*
`maxnodes:`	List of nodes to color red. *(type=list)*
`trim_freq:`	Remove nodes where the frequency is less than this value. *(type=int)*
`layout:`	Graph layout: 'spring', 'random', or 'circular'. *(type=str)*
`wf:`	If True, save the graph to 'graph.pdf' in the current directory. Will overwrite existing files. *(type=bool)*

---

drawGraphRels(*G*, *trim_freq*=0, *layout*='spring', *cols*=['g', 'b', 'r', 'y', 'purple', 'orange'], *labels*=False, *wf*=False, *node_sizing*='freq', *scale*=1.0, *max_size*=500)

Plots a graph to the screen. Colors nodes by their relevance value. Is capable of saving the graph to a PDF file.

Parameters

`G:`	Graph. *(type=NetworkX Graph)*
`trim_freq:`	Remove nodes where the frequency is less than this value. *(type=int)*
`layout:`	Graph layout: 'spring', 'random', or 'circular'. *(type=str)*
`cols:`	Colors to paint nodes. *(type=list)*
`labels:`	Print node labels. *(type=bool)*
`wf:`	If True, save the graph to 'graph.pdf' in the current directory. Will overwrite existing files. *(type=bool)*
`node_sizing:`	'freq', or 'total'. Size nodes on the number of relevant conversations (freq), or total conversations. *(type=str)*
`scale:`	Number by which to scale the node sizes. Might be required for proper display. *(type=float)*
`max_size:`	Limit displayed sizes of nodes to this value. *(type=int)*

---

graphStats(*G*)

Returns summary statistics for a graph.

Parameters

`G`: Graph.
(type=NetworkX Graph)

---

maxNodeFreq(*G*, *freq_type*=`'freq'`)

Calculates the maxnode for a Graph. This is the node with either the highest number of relevant or total conversations on its' adjacent edges.

Parameters

`G`: Graph.
(type=NetworkX Graph)

`freq_type`: Node attribute to calculate: 'freq' or 'total'.
(type=str)

Return Value

Maximum node size in the graph.
(type=int)

---

max_clique(*G*)

Calculates the size of the largest clique in the graph.

Parameters

G : Graph.
(type=NetworkX Graph)

Return Value

The size of the largest clique in the graph.
(type=int)

---

num_of_edges(*G*)

Calculates the number of edges in the graph.

Parameters

G : Graph.
(type=NetworkX Graph)

Return Value

The number of edges in the graph.
(type=int)

---

num_of_nodes(*G*)

Calculates the number of nodes in the graph.

Parameters

G : Graph.
(type=NetworkX Graph)

Return Value

The number of nodes in the graph.
(type=int)

---

pruneGraph(*newG*, *p*)

Trims a graph by pruning all degree one nodes probabilistically.

Parameters

`newG:` Graph to trim.
(type=NetworkX Graph)

`p:` Probability of pruning a degree one node.
(type=float)

Return Value

Trimmed graph.
(type=NetworkX Graph)

---

pruneGraphNodeByDegree(*iG*)

Trims a graph by removing nodes probabilistically by their degree.

Parameters

`iG:` Graph to trim.
(type=NetworkX Graph)

Return Value

Trimmed Graph.
(type=NetworkX Graph)

---

readGraph_CSV(*node_path*, *edge_path*)

Reads a graph from CSV files. Required Node format -> nodename, frequency, relevance, total. Required Edge format -> from, to, ednum, edread, notrel, rel.

Parameters

`node_path`: Filename of node file.
(type=str)

`edge_path`: Filename of edge file.
(type=str)

Return Value

Graph constructed from CSV files.
(type=NetworkX Graph)

---

sij_generator(*num_pijlevels*=2, *file*='sij.csv')

Creates conditional probability tables for Prob(P_ij| S_ij) suitable for import into GraphBuilder.

Parameters

`num_pijlevels`: The number of discrete P_ij levels.
(type=int)

`file`: Filename of output file.
(type=str)

Return Value

Conditional probability table.
(type=list)

---

trimGraphDeep(*iG*, *num_of_nodes*=`10`, *p*=`0.8`, *freq_type*=`'freq'`)

Creates a subset of G. Trims the graph using a "Deep" search pattern. The function first identifies the node with the most relevance (maxnode). All neighbors of the maxnode are added to the graph. From the list of all nodes currently in the graph, the function then determines the node with the next highest relevance, adding its' neighbors to the graph. This process is repeated a specified number of times. All degree one nodes are then trimmed probabilistically.

Parameters

`iG:` Graph.
(type=NetworkX Graph)

`num_of_nodes:` Number of times the algorithm will determine the next node of maximum relevance (rounds).
(type=int)

`p:` Probability of trimming a degree one node.
(type=float)

`freq_type:` 'freq' or 'total' Determines what node attribute to use for graph maxnodes.
(type=str)

Return Value

(Trimmed Graph, List of max_nodes followed).
(type=tuple)

---

trimGraphInfection(*G*, *num_of_nodes*=`300`, *p*=`0.1`, *nzero*=`1`, *freq_type*=`'freq'`)

Creates a subset of G. Trims the graph using an "Infection" method. The function first identifies the node with the most relevance (maxnode). All edges from this

node are then added to the graph with probability p (infected). On the next round of infection, all current edges leading from nodes in the graph are considered for infection. Using this method the graph grows until the node limit is reached.

Parameters

`G:`	Graph. *(type=NetworkX Graph)*
`num_of_nodes:`	Number of desired nodes in the graph. *(type=int)*
`p:`	Probability of infecting neighbors of nodes in the graph. *(type=float)*
`nzero:`	Number of infected nodes at the start of algorithm. *(type=int)*
`freq_type:`	'freq' or 'total' Determines what node attribute to use for graph start point. *(type=str)*

Return Value

Trimmed Graph.
(type=NetworkX Graph)

---

trimGraphWide(*iG*, *num_of_nodes*=`3`, *p*=`0.8`, *freq_type*=`'freq'`)

Creates a subset of G. Trims the graph using a "Wide" search pattern. The function first identifies the node with the most relevance (maxnode). All neighbors of the maxnode are added to the graph. From the list of nodes just added to the graph, the function then determines the node with the next highest relevance, adding its' neighbors to the graph. This process is repeated a specified number of times. All degree one nodes are then trimmed probabilistically.

Parameters

`iG:` Graph.
(type=NetworkX Graph)

`num_of_nodes:` Number of times the algorithm will determine the next node of maximum relevance (rounds).
(type=int)

`p:` Probability of trimming a degree one node.
(type=float)

`freq_type:` 'freq' or 'total' Determines what node attribute to use for graph maxnodes.
(type=str)

Return Value

(Trimmed Graph, List of max_nodes followed).
(type=tuple)

---

writeGraph_CSV(*G*, *node_path*, *edge_path*)

Writes the graph to CSV files. Node format -> nodename, frequency, relevance, total. Edge format -> from, to, ednum, edread, notrel, rel.

Parameters

`G:` Graph.
(type=NetworkX Graph)

`node_path:` Filename of node file.
(type=str)

`edge_path:` Filename of edge file.
(type=str)

APPENDIX C:
Algorithms

C.1 Module Algorithms

A collection of algorithms and support functions that can be run on models created with the GraphBuilderClass.py module.

C.1.1 Functions

FHM(*mod*, *time*, *c*, *depth*, *logfile*='FHMlog.txt', *distances*='FHMdistances.csv', *choice_limit*='null', *func*='_simple_k_nonreduce', *args*=[])

Implements a Finite Depth Markov Decision Process algorithm. Writes detailed results to a log and the distances for each iteration to CSV files. The distances represents p_e* - p_w, or the distance between the p_ij of the optimal edge to screen, and the p_ij of the edge chosen.

Parameters

`mod:`	Graphical model. *(type=GraphBuilder Model)*
`time:`	Max number of items to screen. *(type=int)*
`c:`	Probability of sudden revelation on a screened edge. *(type=float)*
`depth:`	Depth of the Markov Decision Process Tree. *(type=int)*
`logfile:`	Name of output file log. *(type=str)*
`distances:`	Name of distances files. *(type=str)*
`choice_limit:`	Limit the number of edge choices the algorithm takes under consideration. *(type=int)*
`func:`	Function to calculate knowledge gained from a relevant item. *(type=str)*

`args:` List of parameters for reducing function passed in 'func' argument.
(type=list)

Return Value

(The number of relevant items identified, List of distances, Total run time, Average update time).
(type=tuple)

PE(*mod*, *time*, *c*, *logfile*=`'PElog.txt'`, *distances*=`'PEdistances.csv'`, *func*=`'_simple_k_nonreduce'`, *args*=`[]`, *limit*=`'null'`, *snapshot*=`False`, *sres*=`25`)

Implements the Pure Exploitation (PE) algorithm. PE is a greedy algorithm that always chooses an item from the edge with the highest expected probability of being relevant. Ignores exploration. Considered a naive approach. Returns the number of relevant conversations found during the time constraint, as well as a distance list. Writes detailed results to a log and the distances for each iteration to CSV files. The distances represents p_e* - p_w, or the distance between the p_ij of the optimal edge to screen, and the p_ij of the edge chosen.

Parameters

`mod:` Graphical Model.
(type=GraphBuilder Model)

`time:` Max number of items to screen.
(type=int)

`c:` Probability of sudden revelation for an edge.
(type=float)

`logfile:` Name of output log file.
(type=str)

`distances:` Name of distances files.
(type=str)

`func:` Function to calculate knowledge gained from a relevant item.
(type=str)

`args:` List of parameters for reducing function passed in 'func' argument.
(type=list)

`limit:` Only select edge if the number of relevant items already screened from it is less than this value.
(type=int)

`snapshot:` Saves the state of the graph during the algorithm's progression.
(type=boolean)

`sres:` Snapshot interval.
(type=int)

Return Value

(The number of relevant items identified, List of distances, Total run time, Average update time).
(type=tuple)

VDBE(*mod*, *time*, *c*, *delta*, *T*=`0.25`, *inverse_sensitivity*=`0.3`, *logfile*=`'VDBElog.txt'`, *distances*=`'VDBEdistances.csv'`, *func*=`'_simple_k_nonreduce'`, *args*=`[]`)

Implements an algorithm based on the epsilon-greedy Value Difference Based Exploration algorithm. At each iteration the algorithm assigns a probability epsilon that exploration is chosen. Writes detailed results to a log and the distances for each iteration to CSV files. The distances represents p_e* - p_w, or the distance between the p_ij of the optimal edge to screen, and the p_ij of the edge chosen.

Parameters

`mod:` Graphical Model.
(type=GraphBuilder Model)

`time:` Max number of items to screen.
(type=int)

`c:` Probability of sudden revelation on a screened edge.
(type=float)

`delta:` Determines the decay rate of epsilon when the system is stable.
(type=float)

`inverse_sensitivity:` Determines the immediate impact a certain change in expectation has on epsilon.
(type=float)

`logfile:` Name of output file log.
(type=str)

`distances:` Name of distances files.
(type=str)

`func:` Function to calculate knowledge gained from a relevant item.
(type=str)

`args:` List of parameters for reducing function passed in 'func' argument.
(type=list)

Return Value

(The number of relevant items identified, List of distances, Total run time, Average update time).
(type=tuple)

---

conv_Count(*G*)

Calculates the number of items left in the graph available for screening.

Parameters

`G:` Graph.
(type=NetworkX Graph)

Return Value

The number of items available for screening.
(type=int)

---

highest_Pij(*mod*, *func*, *args*)

Finds p_e*, where e* is the edge with unscreened items that has the highest probability of returning a relevant item. This is the true highest value of p_ij for an edge with available items.

Parameters

`mod:` Graphical Model.
(type=GraphBuilder Model)

`func`: Function to calculate knowledge gained from a relevant item.
(type=str)

`args`: List of parameters for reducing function.
(type=list)

Return Value

(Highest true p_ij value, Corresponding edge).
(type=tuple)

---

perfect(*mod*, *time*, *c*, *logfile*=`'perfectlog.txt'`, *func*=`'_simple_k_nonreduce'`, *args*=`[]`)

Implements a greedy algorithm where the true p_ij values are known. Represents a best possible screening method. Returns the number of relevant items found during the time constraint. Writes detailed results to a log.

Parameters

`mod`: Graphical Model.
(type=GraphBuilder Model)

`time`: Max number of items to screen.
(type=int)

`c`: Probability of sudden revelation for an edge.
(type=float)

`logfile`: Name of output log file.
(type=str)

`func`: Function to calculate knowledge gained from a relevant item.
(type=str)

`args`: List of parameters for reducing function passed in 'func' argument.
(type=list)

Return Value

The number of relevant items identified. *(type=int)*

randompick(*mod*, *time*, *c*, *logfile*='randomlog.txt', *distances*='randomdistances.txt', *func*='_simple_k_nonreduce', *args*=[])

Implements a random edge selection method. Represents a worse case scenario. Returns the number of relevant items found during the time constraint. Writes detailed results to a log and the distances for each iteration to CSV files. The distances represents p_e* - p_w, or the distance between the p_ij of the optimal edge to screen, and the p_ij of the edge chosen.

Parameters

`mod`: Graphical Model.
(type=GraphBuilder Model)

`time`: Max number of items to screen.
(type=int)

`c`: Probability of sudden revelation for an edge.
(type=float)

`logfile`: Name of output log file.
(type=str)

`distances`: Name of distances file.
(type=str)

`func`: Function to calculate knowledge gained from a relevant item.
(type=str)

`args`: List of parameters for reducing function passed in 'func' argument.
(type=list)

Return Value

(The number of relevant items identified, List of distances, Total run time, Average update time).
(type=tuple)

reduce_pij_variance(*mod*, *reduce*)

Reduces the variance of the true p_ij values by decreasing the distance of each edge p_ij value to the overall mean p_ij value as a proportion of the current distance.

Parameters

`mod:` Graphical Model.
(type=GraphBuilder Model)

`reduce:` Proportion to reduce distance.
(type=float)

Return Value

Graphical Model with reduced p_ij variance.
(type=GraphBuilder Model)

---

softmax(*mod*, *time*, *c*, *T*, *logfile*=`'SoftMaxlog.txt'`, *distances*=`'SoftMaxdistances.csv'`, *func*=`'_simple_k_nonreduce'`, *args*=`[]`)

Implements the Softmax algorithm. Softmax assigns each edge with a weight that represents the probability an item on the edge is expected to be relevant, and chooses edges to screen items from a distribution built from these weights. Writes detailed results to a log and the distances for each iteration to CSV files. The distances represents p_e* - p_w, or the distance between the p_ij of the optimal edge to screen, and the p_ij of the edge chosen.

Parameters

`mod:` Graphical Model.
(type=GraphBuilder Model)

`time:` Max number of items to screen.
(type=int)

`c:` Probability of sudden revelation for an edge.
(type=float)

`T:` Temperature (0, 1].
(type=float)

`logfile:` Name of output log file.
(type=str)

`distances:` Name of distances files.
(type=str)

`func:` Function to calculate knowledge gained from a relevant item.
(type=str)

`args:` List of parameters for reducing function passed in 'func' argument.
(type=list)

Return Value

(The number of relevant items identified, List of distances, Total run time, Average update time).
(type=tuple)

C.2 Module ChoiceNode

Creates a Choice Node Object.

C.2.1 Class ChoiceNode

The ChoiceNode class supports the creation of a ChoiceNode. ChoiceNodes are used in support of Finite Depth (FHM) algorithms. Utilizes the RandomNode.py module.

Methods

__init__(*self*, *GB*, *depth*, *rounds_remaining*, *choice_limit*='null', *func*='null', *args*=[])

Construct a ChoiceNode. The FHM alorithm can be initiated by creation of a ChoiceNode and subsequent calling of its getVal() method.

Parameters

`GB:`	GraphBuilder. *(type=GraphBuilder Object)*
`depth:`	Depth of the MDP tree (how far to look into future). *(type=int)*
`rounds_remaining:`	The number of screening rounds remaining. *(type=int)*
`choice_limit:`	Limit the number of RandomNodes to create. *(type=int)*
`func:`	Function to calculate knowledge gained from a relevant item. *(type=str)*
`args:`	List of parameters for reducing function passed in 'func' argument. *(type=list)*

getVal(*self*)

Returns the value of the ChoiceNode.

Return Value

(Best edge choice, Expected value of the choice).
(type=tuple)

C.3 Module RandomNode

Creates a RandomNode Object.

C.3.1 Class RandomNode

The RandomNode class supports the creation of a RandomNode. RandomNode is used in support of the Finite Depth (FHM) algorithm. Requires the ChoiceNode.py module.

Methods

__init__(*self*, *GB*, *edge*, *depth*, *rounds_remaining*, *choice_limit*=`'null'`, *func*=`'null'`, *args*=`[]`)

Construct a RandomNode Object.

Parameters

`GB:`	Object of type GraphBuilder. *(type=GraphBuilder Object)*
`edge:`	Edge of choice. *(type=tuple)*
`depth:`	Depth of the MDP tree (how far to look into future). *(type=int)*
`rounds_remaining:`	The number of screening rounds remaining. *(type=int)*
`choice_limit:`	Limit the number of RandomNodes to create. *(type=int)*
`func:`	Function to calculate knowledge gained from a relevant item. *(type=str)*
`args:`	List of parameters for reducing function passed in 'func' argument. *(type=list)*

getVal(*self*)

Returns the expected value of the RandomNode.

Return Value

Maximum expected value.
(type=float)

APPENDIX D: GraphBuilderNaive

D.1 Module GraphBuilderNaive

Extends the GraphBuilderClass module by building a naive model with no correlation information.

D.1.1 Class GraphBuilder

The GraphBuilder class supports the creation of a naive graphical model, and accompanying support functions required to test various intelligence collection algorithms. Specific algorithms can be found in the Algorithms.py module.

Methods

__init__(*self*, *G*, *joint_prob_prefix*=`'joint'`, *pij_dij_file*=`'pij_dij.csv'`, *sij_file*=`'sij.csv'`, *c*=`0.5`, *precision*=`5`)

Construct a naive Graphical Model by reading in NetworkX graph and accompanying probability distributions.

Parameters

`G:`	Graph to construct graphical model from. *(type=NetworkX Graph)*
`joint_prob_prefix:`	Prefix of file names that contain the joint distribution of the D_i's. *(type=int)*
`pij_dij_file:`	Filename for conditional probability distribution of P_ij, given, D_i, D_j. *(type=str)*
`sij_file:`	Filename for probability of P_ij, given S_ij. *(type=str)*
`precision:`	Number of digits to display in conditional probability tables. *(type=int)*

Return Value

Graphical model object.

count_remaining(*self*)

Calculates the remaining items available for screening in the model.

Return Value

The number of items available for screening.
(type=int)

edge_update(*self*, *edge*, *value*, *sumout*=`True`)

Update the graphical model after screening an item.

Parameters

`edge:` Edge to update.
(type=tuple)

`value:` Value of edge update.
(type=int)

`sumout:` If True, sum out the S_ij factor after update.
(type=bool)

expected_di(*self*, *node*)

Displays the marginal probability distribution for a node.

Parameters

`node:` Graph node.
(type=str)

Return Value

Dictionary of probabilities.
(type=dict)

---

expected_pij(*self*, *edge*, *limit*='null', *args*=[])

Calculates the expected P_ij for a requested edge.

Parameters

`edge`: Graph edge.
(type=tuple)

`limit`: Name of knowledge limiting function, if specified.
(type=str)

`args`: A list of knowledge limit function arguments.
(type=list)

Return Value

The expected P_ij for the requested edge.
(type=float)

---

highest_expected_pij(*self*, *numEdges*=None, *limit*='null', *args*=[])

Generates a list of edges sorted from highest to lowest expected probability for a relevant item.

Parameters

`numEdges`: Length of list to return.
(type=int)

`limit`: Name of knowledge limiting function, if specified.
(type=str)

`args`: A list of knowledge limit function arguments.
(type=list)

Return Value

Descending list of expected P_ij values in tuple form (Edge, Expected P_ij).
(type=list)

node_update(*self*, *node*, *value*)

Update a node relevance value from sudden revelation.

Parameters

`node`: Node to update.
(type=str)

`value`: Value of revelation.
(type=str)

random_draw(*self*, *edge*)

Computes a random draw on an edge using the true p_ij value, and returns the relevance value.

Parameters

`edge`: Edge on which to perform a random item draw.
(type=tuple)

Return Value

Relevance value of the item.
(type=int)

sudden_relevance_simple(*self*, *node*, *c*)

Computes the results of a sudden revelation realization on a node. Relevance is calculated with a fixed probability parameter.

Parameters

`node`: Node on which to perform a sudden revelation check.
(type=str)

`c`: Probability of sudden revelation on the node.
(type=float)

Return Value

(Boolean value for whether sudden revelation realization occurred, The node for which any sudden revelation occurred, The value of the revelation).
(type=tuple)

THIS PAGE INTENTIONALLY LEFT BLANK

REFERENCES

Atkinson, M. P., L. M. Wein. 2010. An overlapping networks approach to resource allocation for domestic counterterrorism. *Studies in Conflict & Terrorism* **33**(7) 618–651.

Berry, D. A., B. Fristedt. 1985. *Bandit Problems*. Chapman and Hall, MI.

Daw, N. D., J. P. O'Doherty, P. Dayan, B. Seymour, R. J. Dolan. 2006. Cortical substrates for exploratory decisions in humans. *Nature* **441**(7095) 876–879.

Deitchman, S. J. 1962. A lanchester model of guerrilla warfare. *Operations Research* **10**(6) 818–827.

Diesner, J., K. M. Carley. 2005. Exploration of communication networks from the Enron email corpus. *SIAM International Conference on Data Mining: Workshop on Link Analysis, Counterterrorism and Security*. Citeseer, Newport Beach, CA.

Frazier, P., W. Powell, S. Dayanik. 2009. The knowledge-gradient policy for correlated normal beliefs. *INFORMS Journal on Computing* **21** 591–613.

Fu, M. C., J. Q. Hu, C. H. Chen, X. Xiong. 2007. Simulation allocation for determining the best design in the process of correlated sampling. *INFORMS Journal on Computing* **19** 101–111.

Hedley, J. H. 2007. Analysis for strategic intelligence. L.K. Johnson, ed. *Strategic Intelligence: understanding the hidden side of government*. Praeger, Santa Barbara, CA.

Kampstra, P. 2008. Beanplot: A boxplot alternative for visual comparison of distributions. *Journal of Statisical Software* **28**.

Kaplan, E. H. 2012. OR forum—intelligence operations research: The 2010 Philip McCord morse lecture. *Operations Research* **60**(6) 1297–1309.

Koller, D., N. Friedman. 2009. *Probabilistic Graphical Models: Principles and Techniques (Adaptive Computation and Machine Learning Series)*. 1st ed. The MIT Press, Cambridge, MA.

Nevo, Y. 2011. Information selection in intelligence processing. Master's thesis in operations research, Naval Postgraduate School, Monterey, CA.

Pearl, J. 1986. Fusion, propagation and structuring in belief networks. *Artificial Intelligence* **29** 241–288.

Schaffer, M. B. 1968. Lanchester models of guerrilla engagements. *Operations Research* **16**(3) 457–488.

Thrun, S. B. 1992. *Handbook of intelligent control: Neural, fuzzy and adaptive approaches*, chap. The role of exploration in learning control. Van Nostrand Reinhold, Florence, KY, 527–559.

Tokic, M., G. Palm. 2011. Value-difference based exploration: adaptive control between epsilon-greedy and softmax. *KI 2011: Advances in Artificial Intelligence*. Springer-Verlag, Berlin, Germany, 335–346.

Zlotnik, J. 1967. A theorem for prediction. *Studies in Intelligence* **11** 1–2.

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudly Knox Library
   Naval Postgraduate School
   Monterey, California